Tree Based Hierarchical Reinforcement Learning

William T. B. Uther

August 2002

CMU-CS-02-169

Department of Computer Science
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the United States Air Force under Grants Nos. F30602-00-2-0549 and F30602-98-2-0135.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **AUG 2002**	2. REPORT TYPE	3. DATES COVERED **00-00-2002 to 00-00-2002**
4. TITLE AND SUBTITLE **Tree Based Hierarchical Reinforcement Learning**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**163**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: Reinforcement Learning, Regression Trees, Markov Decision Processes, Semi-Markov Decision Processes, Constructive Induction, Hierarchical Reinforcement Learning

Abstract

In this thesis we investigate methods for speeding up automatic control algorithms. Specifically, we provide new abstraction techniques for Reinforcement Learning and Semi-Markov Decision Processes (SMDPs). We introduce the use of policies as temporally abstract actions. This is different from previous definitions of temporally abstract actions as we do not have termination criteria. We provide an approach for processing previously solved problems to extract these policies. We also contribute a method for using supplied or extracted policies to guide and speed up problem solving of new problems. We treat extracting policies as a supervised learning task and introduce the Lumberjack algorithm that extracts repeated sub-structure within a decision tree. We then introduce the TTree algorithm that combines state and temporal abstraction to increase problem solving speed on new problems. TTree solves SMDPs by using both user and machine supplied policies as temporally abstract actions while generating its own tree based abstract state representation. By combining state and temporal abstraction in this way, TTree is the only known SMDP algorithm that is able to ignore irrelevant or harmful subregions within a supplied abstract action while still making use of other parts of the abstract action.

Acknowledgments

This work would not have been possible without the support of many people. Foremost amongst these is my advisor, Manuela, who showed me how to draw useful lessons from many research paths. My family also deserves much credit for their support during my time in the United States; it is wonderful knowing they're always behind me. Many thanks to my friends for keeping me, if not sane, then at least from raving too much. The denizens of 6D12 all helped; Sammie (an excellent turler), Susan, Lindsey and Krissie have all been wonderful friends. My Australian friends, Cameron, Sarah and Max gave support from all corners of the world; maybe we'll all be together once more... I am also grateful for the improv troupes I performed with. Both Friday Nite Improvs and the No Parking Players gave me insanity in play to match the insanity at work. There are many other friends who helped in their own way; there are too many to mention, but I am particularly indebted to Lakshmi, Chris, Alanna and Maria. Finally, to my office mates and others in the research group with whom I had some wonderful discussions and much fun; Elly, Gal, Brett and Rick, my thanks.

Contents

List of Figures

List of Tables

List of Procedures

Chapter 1

Introduction

There is an important class of control problems known to computer scientists as *Reinforcement Learning* (*RL*) tasks (Sutton and Barto, 1998; Watkins and Dayan, 1992; Kaelbling et al., 1996). This class of tasks is flexible enough to contain a wide range of challenges for robotics. Examples of challenges that can be formulated as reinforcement learning tasks include a robot learning to walk, automatic piloting of a helicopter, games such as Pente and the Towers of Hanoi, and warehouse scheduling.

In reinforcement learning tasks we assume that an agent starts with very little knowledge of its environment. It knows neither the dynamics of the world, nor a reward structure for its actions. The agent discovers the dynamics and reward structure through experimentation. It performs actions in the world and observes both the results of the actions and the reward it gets for the actions. The agent's goal is to learn which actions to perform such that it maximizes the reward it receives over time.

One important formalism for reinforcement learning is the Semi-Markov Decision Process (SMDP). Defined more completely in Chapter 2, this is a formal description of one way to specify a set of world dynamics and a reward structure. The world is assumed to be a form of stochastic state machine with the agent having some control over the transition probabilities. With this formalism the Reinforcement Learning problem can be broken into two problems. The first is to learn a model of the world as an SMDP. The second is to use the SMDP model of the world to discover a policy that maximizes the agent's reward. A *policy* specifies which action is performed in each world state.

Often the number of discrete states of the world is very large, and this can make finding a policy difficult. In particular, often the state of the world is specified in terms of state variables and the number of individual world states is exponential in the number of state variables. If we need to consider each state separately in the policy, for example if we are representing the policy using a large table, then we need to do an exponential amount of work in the number of state variables and this is slow. To speed things up, we exploit structure in the policy.

We focus in this thesis on structure related to temporal abstraction – sequences of actions that are repeated within and between problem solutions. In the process of achieving their goals, most agents have to repeat similar subtasks multiple times. If the methods of performing these subtasks can be stored and re-used when needed, then the work of finding how to perform a subtask only has to be done once.

As actions cause the agent to move through the world, temporal patterns in the actions of an agent become spatial patterns in the policy. We discover these patterns by looking for substructure repetition within a tree representation of the policy. We use syntactic matching within policies to recognize the repeated action sequences that are formed when solving a subtask multiple times.

Since finding these repeated patterns within a policy *as you are learning it* is a difficult problem, the approach we use in this thesis is to look within previously learned policies. For example, assume we have a series of similar problems to solve. We solve the first problem in the series using standard techniques without any information from prior tasks. Having solved this problem, we can now process the policy we have found, to look for repeated substructure. If a structure is found repeatedly in the solution to one problem then it is likely that it appears in the solution to other problems. Having found these repeated problem pieces, we then use them to help solve further problems.

Let us look at a more concrete example. Imagine we have a robot, Robbie, with two legs; a left leg and a right leg. With a few restrictions, each of these legs can be raised and lowered one unit, and the raised foot can be moved one unit in each of the four compass directions: north, south, east and west. The legs are restricted in movement so that they are not both in the air at the same time. They are also restricted to not be diagonally separated, *e.g.* , the right leg can be either east or north of the left leg, but it cannot be both east and north of the left leg.

More formally, we represent the position of the robot using the two dimensional coordinates of the right foot, $\langle x, y \rangle$. We then represent the pose of the robot's legs by storing the three dimensional position of the left foot relative to the right foot, $\langle \Delta x, \Delta y, \Delta z \rangle$. We represent East on the $+x$ axis, North on the $+y$ axis and up on the $+z$ axis. The formal restrictions on movement are that Δx and Δy cannot both be non-zero at the same time and that each of Δx, Δy and Δz are in the set $\{-1, 0, 1\}$. Consider some example leg poses; if both feet are on the ground with the left foot to the north of the right foot, then the pose of the legs would be represented by the tuple $\langle 0, 1, 0\rangle$. $\langle 0, -1, 0\rangle$ represents the left foot being south of the right foot. If the legs are together, but the right foot is raised, we would represent that as $\langle 0, 0, -1\rangle$. If that raised right foot were moved north, then the robot is considered to have moved north one square, and the left foot has moved one unit south relative to the right foot. The resulting pose of the legs is $\langle 0, -1, -1\rangle$.

To help get a feel for this domain, let us consider a 'simple' policy. Table 1.1 shows a policy for walking north starting with both feet together on the ground. We only show the Δz-Δy plane as that is all that is needed when you start with your feet together. This policy is 'simple' in that it completely ignores the global location of the robot and any maze it might be in; it just walks north in an open area. Notice that even this 'simple' policy is not trivial. Also note that the state space for Robbie is simpler than that used by real legged robots; *e.g.* the Sony AIBO robot requires three joint angles to be specified for each of four legs and 'simple' policies like walking are even more complex for this style of robot.

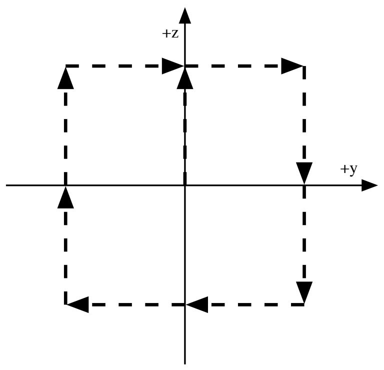
Now let us see how the approach taken in this thesis applies to Robbie solving the series of problems shown in Figure 1.1. Each problem is a maze, and Robbie's goal in each problem is to walk through the maze to reach the grey square. The walls limit Robbie's ability to move his legs – any action that would cause Robbie to end up with his feet on either side of a wall fails, and gives Robbie a small negative reward.

The first problem, shown in Figure 1.1a, is solved without the benefit of any information from prior tasks. A simplified representation of Robbie's solution to the first problem is shown in Figure 1.2a. The solution shown has been simplified by abstracting away the details of the walking motion and just leaving the direction Robbie walks in each part of the maze. In the full policy, each area where Robbie walks north contains a piece of policy as complex as the policy shown in Table 1.1. Walking in other directions is similarly complex. Now let us consider solving the second problem under two scenarios. In the first scenario, Robbie solves the problem in Figure 1.1b the traditional way, without the benefit of any information from prior tasks. In the second scenario, Robbie processes the solution to the first task and uses the information learned to help solve the second task. In either case, the final policy learned is that shown in Figure 1.2b.

In the first case, Robbie has to learn the policy shown in Figure 1.2b from scratch. This means that Robbie has to learn to walk north again. Moreover, as there is no transfer

```
if $\Delta z = 0$ then {both feet on the ground}
   if $\Delta y > 0$ then {left foot north of right foot}
      raise the right foot
   else
      raise the left foot
   end if
else if $\Delta z = 1$ then {the left foot is in the air}
   if $\Delta y > 0$ then {left foot north of right foot}
      lower the left foot
   else
      move the raised foot north one unit
   end if
else {the right foot is in the air}
   if $\Delta y < 0$ then {right foot north of left foot}
      lower the right foot
   else
      move the raised foot north one unit
   end if
end if
```

(a)



(b)

Table 1.1: The policy for walking north when starting with both feet together. (a) Shows the policy in tree form, (b) shows the policy in diagram form. Note: only the Δz-Δy plane of the policy is shown as that is all that is required when starting to walk with your feet together.

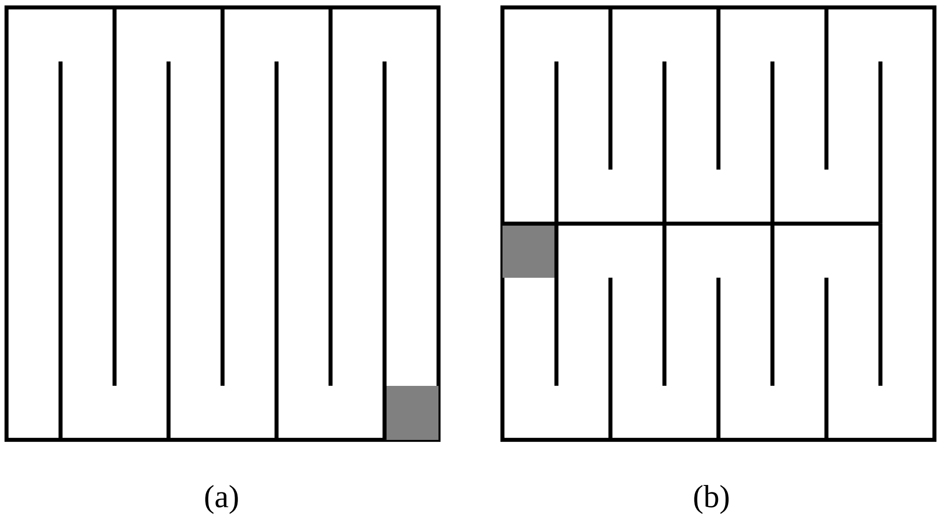


Figure 1.1: A pair of example mazes for our robot. The robots's goal in each maze is to reach the grey patch.

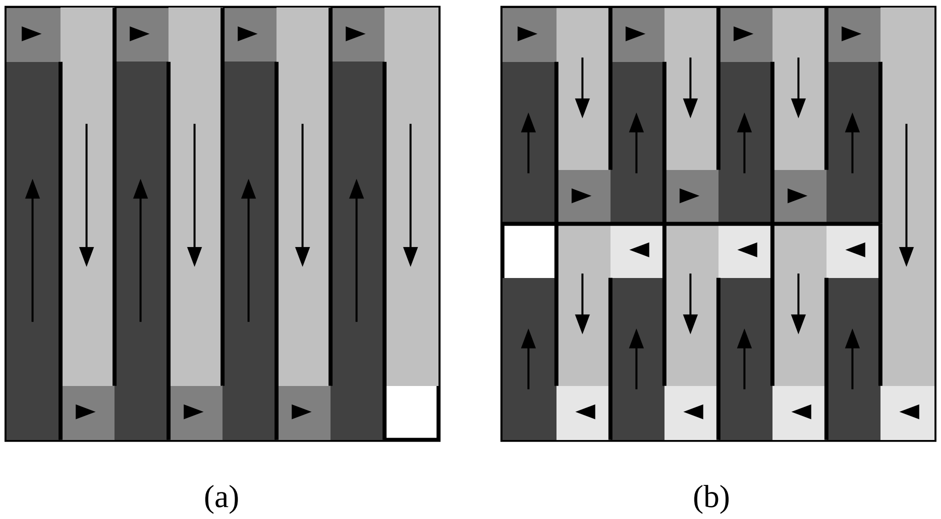


Figure 1.2: The abstract policies Robbie finds for the mazes in Figure 1.1. Only the two dimensions of the state space that describe Robbie's global location are shown for each maze. The other dimensions are abstracted into a direction robbie must walk. In full detail, each region marked contains a sub-policy at least as complex as the one shown in Table 1.1. (a) The solution to the maze shown in Figure 1.1a. (b) The solution to the maze shown in Figure 1.1b.

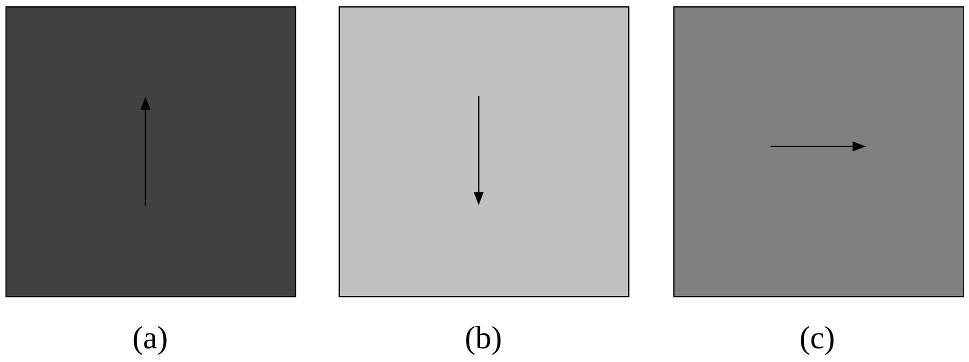


Figure 1.3: Three policies extracted from Figure 1.2a. As in Figure 1.2, only the $\langle x, y\rangle$ plane is shown. When abstracted to just this plane these policies are degenerate – they are uniform over $\langle x, y\rangle$ plane. (a) The policy for walking north. (b) The policy for walking south. (c) The policy for walking east. Non-abstract representations of these policies are not degenerate, and are shown in Tables 1.1 and 1.2.

between subtasks within a problem, Robbie has to learn to walk north eight times while solving this maze. Additionally, he has to learn to walk south, east and west seven times each. Remember that learning each of these walking subtasks is a nontrivial amount of work.

In the second case, using the approach proposed in this thesis, Robbie does not attack the problem directly. Robbie first processes his solution to the first problem looking for non-trivial repeated sub-policies within that policy (Robbie's solution to the first problem is shown in Figure 1.2a). The repeated sub-policies are extracted from their locations in the old policy and generalized to cover the entire state space. The result is a set of policies; functions that return an action to perform for every state. The three policies we would expect to be extracted from Robbie's solution to the first problem are shown in Figure 1.3. These are the policies for walking north, south and east. We would not expect to generate a policy for walking west, as there is no example of this in the original policy.

Now Robbie can start solving his second problem using knowledge extracted from the first problem. Robbie no longer has to learn to walk north from scratch, rather he tests the walk-north policy and finds that it moves him north. Instead of having to learn to walk north 8 times, Robbie has to test the policies he has found eight times. This testing of already known policies is a much more efficient process than solving the problem from scratch, and so Robbie is able to solve the second problem more easily than the first. In some cases, Robbie does not have a policy from a previous problem that helps. In particular, there is no place where Robbie has to walk west in the first problem Robbie encountered, and hence no policy for walking west is available when solving the second problem. Robbie has to learn to walk west from scratch in the second problem, and as there is no reuse while solving a problem, the sub-policy of walking west is learned seven times.

1.1 Thesis Problem and Contributions

The question I investigate in this thesis is:

> How can finding a policy for one Semi-Markov Decision Problem be used to speed up the discovery of a policy for other, similar, Semi-Markov Decision Problems?

The approach outlined in this thesis is to use a representation of the policy that allows symbolic analysis: we use a tree based representation. The thesis question is then approached by breaking the problem down into two related sub-questions. How can useful information be extracted from a previously solved problem? And, how can the information extracted be used to help solve new problems? We then proceed to answer these two questions separately.

1.1.1 Extracting useful information from solved problems

The first part of extracting useful information is to decide on what information may be useful in future problems. We detect repeated temporal patterns in the behavior of the robot. If they are repeated in one problem then we assume they will be repeated in future problems.

As the world our agent moves through is fully observable, temporal patterns in the behavior of the robot become spatial patterns in the policy. To see this, consider Table 1.1b which shows the policy for walking north laid out graphically. Described as a series of actions, starting with both feet on the ground and the right foot north of the left, we get: lift left foot, move raised foot north, move raised foot north, lower left foot, raise right foot, move raised foot north, move raised foot north, lower right foot. This sequence repeats for as long as we walk north. Because each of these actions moves us to a new state, the sequence of actions lays out a trajectory through the state space, and the temporal pattern forms a corresponding spatial pattern.

```
if $\Delta z = 0$ then {both feet on the ground}
    if $\Delta y <= 0$ then
        raise the left foot
    else
        raise the right foot
    end if
else if $\Delta z = 1$ then {the left foot is in the air}
    if $\Delta y < 0$ then
        lower the left foot
    else
        move the raised foot south one unit
    end if
else {the right foot is in the air}
    if $\Delta y > 0$ then
        lower the right foot
    else
        move the raised foot south one unit
    end if
end if
```

(a)

```
if $\Delta z = 0$ then {both feet on the ground}
    if $\Delta x <= 0$ then
        raise the left foot
    else
        raise the right foot
    end if
else if $\Delta z = 1$ then {the left foot is in the air}
    if $\Delta x > 0$ then
        lower the left foot
    else
        move the raised foot east one unit
    end if
else {the right foot is in the air}
    if $\Delta x < 0$ then
        lower the right foot
    else
        move the raised foot east one unit
    end if
end if
```

(b)

Table 1.2: The policies for walking (a) south and (b) east

Our approach is to use supervised learning detect these patterns. We use a tree-based supervised learning system to re-represent the policy by re-learning it. During this learning process we process the structure of the tree to detect repeated spatial patterns.

A *supervised learning* problem is one where we are supplied with data as a set of input/output pairs, assumed to have been sampled from an unknown function, and we wish to reconstruct the function that generates the data. To solve a supervised learning problem, a program must process the supplied examples to find a function that accurately predicts the output for a given input vector. In our case the function is the policy from a previous problem that is supplied to the learner as a series of state/action pairs.

Most supervised learning techniques attempt to increase the accuracy of their function approximation through generalization: for ‘similar’ inputs, the outputs are assumed to be ‘similar’. The traditional definition of similar is based upon distance in input space: ‘nearby’ states are assumed to be ‘similar’. The form of generalization that this definition of similar generates is already handled by state abstraction techniques in reinforcement learning. We want a definition of similar that helps us find spatial patterns.

In this thesis we use a definition of similar based upon structural similarity within the tree representation in addition to the normal ‘nearby in input space’ measure. Consider Robbie’s solution to the first maze represented in Figure 1.2a. This policy is represented as a tree in Table 1.3. Notice that many of the sub-trees are identical, and in the table are simply references to other tables.

By detecting this repeated internal structure in the representation of the policy, we can extract some repeated patterns in the policy. Extracting a subtree means removing the surrounding tree structure. This surrounding structure is a limitation on when the sub-tree supplies the policy for the agent. Removing this surrounding structure generalizes the sub-tree so it can apply in the entire state space.

1.1.2 Using policies to speed up problem solving

We wish to use policies as abstract actions to improve the speed at which good solutions are found. We want to do this without limiting our algorithm’s convergence to optimality in the limit of infinite exploration. Our approach is to extend the set of supplied policies so that any policy can be represented as a combination of the extended set of policies, and then to look for the best combination.

We extend the set of supplied policies by adding a set of degenerate policies, one for

```
if $x < 1$ then
  if $y < 9$ then
    Insert the walking north policy tree here. See Table 1.1a for the complete subtree.
  else
    Insert the walking east policy tree here. See Table 1.2b for the complete subtree.
  end if
else if $x < 2$ then
  if $y < 1$ then
    Insert the walking east policy tree here. See Table 1.2b for the complete subtree.
  else
    Insert the walking south policy tree here. See Table 1.2a for the complete subtree.
  end if
else if $x < 3$ then
  if $y < 9$ then
    Insert the walking north policy tree here. See Table 1.1a for the complete subtree.
  else
    Insert the walking east policy tree here. See Table 1.2b for the complete subtree.
  end if
else if $x < 4$ then
  if $y < 1$ then
    Insert the walking east policy tree here. See Table 1.2b for the complete subtree.
  else
    Insert the walking south policy tree here. See Table 1.2a for the complete subtree.
  end if
else
  The rest of the state space is similar to the above, and we leave it out to save space.
end if
```

Table 1.3: The policy, represented as a tree, for solving the maze in Figure 1.1a. This is the same policy represented diagrammatically in Figure 1.2a.

each action. These policies perform the same action in every state. Policies are then combined by using a tree structure to map regions of state space to policies. In this way it is possible to combine the set of degenerate policies to form any other policy. Adding the set of policies found from previous problem solving episodes makes it faster to find some combinations. We prove, under some assumptions given in Chapter 5, that our TTree algorithm finds a combination that optimizes the reward obtained by the policy.

1.1.3 Contributions

In summary, we present in this thesis i) an algorithm for automatically extracting repeated substructure in the policies of previously solved problems, ii) an algorithm that can use policies as abstract actions to solve related problems more efficiently than without the abstract actions, and finally, iii) formal machinery that allows us to prove our polling style algorithm is correct.

1.2 Thesis Overview

Chapter 2 is an introduction to the formal framework used throughout this thesis. Reinforcement Learning and the related Semi-Markov Decision Processes (SMDPs) are formally described. Additionally, a number of different ways of representing information within the SMDP framework are described. Finally, we introduce some of the experimental domains used in the thesis.

Chapter 3 describes a tree-based algorithm for solving reinforcement learning problems. This algorithm demonstrates many of the techniques for using decision/regression trees in reinforcement learning, but cannot make effective use of abstract actions.

Chapter 4 describes the Lumberjack algorithm. This algorithm is our approach to the first problem: extracting a behavior that can be reused. The Lumberjack algorithm does not completely solve this problem. Rather, it is a supervised learning algorithm that can be used to generate a decomposed representation of any function. Some manual intervention is required to use it to decompose policies.

Chapter 5 introduces a second reinforcement learning algorithm, TTree, which can make use of policies as abstract actions to improve its problem solving speed.

Chapter 6 describes related work on using macros for reinforcement learning.

Chapter 2

Technical Overview

As described in the previous chapter, this thesis presents efficient algorithms for Reinforcement Learning and Semi-Markov Decision Problem solving (Puterman, 1994; Sutton and Barto, 1998). This chapter formally describes SMDPs, some standard solution methods and theory for SMDPs, and then describes some of the domains we use to evaluate our algorithms.

2.1 Semi-Markov Decision Processes

We are interested in control for reinforcement learning problems. In a reinforcement learning problem, an agent[1] initially knows nothing about the world. The agent moves about the world and learns both the world dynamics and the reward structure of the world (either implicitly or explicitly). The goal of our algorithms is to find a set of behaviors that maximizes the sum of rewards received by the agent as it moves about the world. In order to formalize this process, we assume that a model for both the world dynamics and the reward structure is learned explicitly. The properties of this combined model are formalized here as a *Semi-Markov Decision Process* (*SMDP*).

Before introducing SMDPs we briefly describe a simplified model, the *Markov Decision Process* (*MDP*). An MDP is a type of state-machine. It has a collection of states with transitions between them. In a normal finite state machine the resulting state of a transition is dependent upon the starting state of the transition. In an MDP, there is also an agent.

[1]Here we use the term agent to refer to a control mechanism in the broadest sense, from extremely simple to very complex.

The resulting state for a transition depends not only upon the starting state, but also upon an action performed by the agent. In this way, the effects of an agent upon the world can be modelled. An MDP also includes a model of the reward received by the agent for each transition.

Use of a Markov Decision Process implies certain assumptions about the world. We assume that the state contains all relevant information about the robot's history. This assumption is known as the *Markov* assumption. Given the current state, information about previous states the robot has passed through tells us nothing. In particular, it tells us nothing about the effects of actions nor about the rewards for those actions. For an office robot, the state would include such information as the location of the robot, the current position of any robot actuators as well as less obvious information such as whether the coffeepot in the room down the hall has fresh coffee. Every relevant piece of information about the world is contained in the state.

A Semi-Markov Decision Process is the same as a Markov Decision Process, but transitions can take varying amounts of time. This allows us to model a series of transitions through one SMDP as a single transition in a more abstract SMDP.

Formally, an SMDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$.[2] $\mathcal{S}$ is the set of states. We use lower case s to refer to particular states, *e.g.* $s \in \mathcal{S}, s' \in \mathcal{S}$. When referring to a particular transition in the SMDP, we generally use s to refer to the start state of the transition and s' to refer to the resulting state. We assume that the states embed into an n-dimensional space: $\mathcal{S} \equiv \mathcal{S}_1 \times \mathcal{S}_2 \times \mathcal{S}_3 \times \cdots \times \mathcal{S}_n$. $\mathcal{A}$ is the set of actions. We use lower case a to refer to particular actions, *e.g.* $a_i \in \mathcal{A}$. When discussing the algorithms formally we assume $\mathcal{A}$ is constant across all states. The extension to have $\mathcal{A}$ depend upon the current state is straightforward.

$P_{s,a}(s', t) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \Re \to [0, 1]$ is a joint probability distribution over both next-states and transition times, defined for each state/action pair. It is the inclusion of time in this distribution that separates an SMDP from an MDP. $R(s, a) : \mathcal{S} \times \mathcal{A} \to \Re$ defines the expected reward for performing an action in a state. In general, the reward for an individual transition can be stochastic and depend upon both the resulting state and the time for the transition. However we only need the expected reward, so any stochasticity drops out. Additionally, as the resulting state and the time for the transition depend only upon the state and action, they also fall out of the expectation.

[2] P is often known as $\mathcal{Q}$ in the SMDP literature, however as we are using Q elsewhere, we are using MDP notation to avoid confusion.

When our agent is first switched on, the world is in a particular state, s. We assume that the agent always knows the current state of the world; *i.e.* that the world is *fully observable*.[3] The agent then performs an action, a. That action takes a length of time, t, to move the agent to a new state, s', the time and resulting state determined by $P_{s,a}(s', t)$. The agent gets reward for the transition with expected value $R(s, a)$.

Some domains have an additional set of states known as *absorbing states*. These states have no transitions leading out of them. When one of these states is entered the agent can do nothing and receives no further reward.

We impose the constraint on P that there is no probability of a negative time transition: $\forall s, a, s'$ and $t < 0$, $P_{s,a}(s', t) = 0$. Also, we require that there are no zero-time, probability 1 cycles in the state space. More formally, there is no policy, $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$, and state s, such that when you start in state s the set of possible trajectories all take zero time and lead back to s. Formally, there does not exist an assignment of actions to the states in $\mathcal{S}$, $\pi(s)$, such that there exists a start state, s, and a set of m trajectories from s back to s, $\big\{\{s^0_0, s^0_1, \ldots, s^0_{n_0}\}, \{s^1_0, s^1_1, \ldots, s^1_{n_1}\}, \cdots, \{s^m_0, s^m_1, \ldots, s^m_{n_m}\}\big\}$, where, the sum of the probability that those trajectories take no time is 1:

$$\sum_{j=0}^{m} \left[P_{s,\pi(s)}(s^j_0, 0) P_{s^j_{n_j},\pi(s^j_{n_j})}(s, 0) \prod_{i=0}^{n_j-1} P_{s^j_i,\pi(s^j_i)}(s^j_{(i+1)}, 0) \right] = 1 \tag{2.1}$$

The usual way to avoid these zero-time cycles is to have no zero-time transitions. This is the approach we take in this thesis.

2.1.1 Policies: controlling SMDPs

The reason we model the world as an SMDP is to help the agent learn a policy. A policy is a method of controlling an agent in an environment, see Puterman (1994) for a complete taxonomy. In this thesis we concentrate on a subset of general policies: complete, deterministic, Markovian policies. Unless stated otherwise we will drop the qualifiers and just use the term ‘policy’.

A complete, deterministic, Markovian policy, $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$, is a function from states to actions. For each state, the policy returns the action the agent performs in that state. In

[3]Much work has been done with partially observable worlds, where only some information is known about the current state. While partial observability is a very important research area we do not deal with it in this thesis.

particular, we want the policy that maximizes the reward for the agent.

It is known that deterministic Markovian policies represent a useful subclass of general policies, however they are not the only type of policy. We can define a *stochastic policy* that maps states to a probability distribution over actions. We can also define a non-Markovian policy that maps the entire history of the agent into an action. For MDPs and SMDP we know that there exist optimal, deterministic, Markovian policies and hence we can generally ignore other classes of policies.

As stated above, our optimality criterion is to maximize the sum of rewards. We refer to a policy that optimizes the sum of rewards as π^*. In the following sections, we formalize the concept of a trajectory through the state space and use that to formalize this concept of 'sum of rewards'.

2.1.2 Trajectories through the world

The agent starts at a particular state, s_0.[4] It then follows a series of transitions leading it through a series of states: $\langle s_0, s_1, s_2, \ldots \rangle$. This series of states is the agent's *trajectory*. If there are any absorbing states then the trajectory is defined to end once the agent enters an absorbing state.

Additionally, the transitions between states take varying amounts of time. We let t_i be the time taken by the ith transition, and τ_i be the time at the start of the ith transition. These are related in the obvious way:

$$\tau_i = \sum_{j=0}^{i-1} t_j \tag{2.2}$$

We can list the states in the trajectory annotated with the time the agent arrives in that state: $\langle s_0^{\tau_0}, s_1^{\tau_1}, s_2^{\tau_2}, \ldots \rangle$. Finally, we can list the rewards gathered by the agent, again annotated with the time they were gathered: $\langle r_0^{\tau_0}, r_1^{\tau_1}, r_2^{\tau_2}, \ldots \rangle$.

[4]Sometimes agents are defined to have a starting distribution over states, rather than a starting state. The two formulations are equivalent. A starting distribution can be transformed into a single start state by adding a new start state and making the transitions out of that state follow the starting distribution.

2.1.3 Optimality criteria

For a particular trajectory of the agent we can find the sum of rewards, SoR, gathered by the agent:

$$\text{SoR} = \sum_{i=0}^{\infty} r_i \tag{2.3}$$

However, the trajectory followed by the agent is stochastic: given the same SMDP and policy, different experimental trials result in different trajectories. To account for this, we alter our criterion to maximize the *expected* sum of rewards, ESoR, over all possible trajectories.

$$\text{ESoR} = \mathop{\mathrm{E}}_{\text{trajectories}}\left[\sum_{i=0}^{\infty} r_i\right] \tag{2.4}$$

where $\mathrm{E}_{\text{trajectories}}[X]$ is the expected value of X over all trajectories.

Unfortunately there is another issue. This infinite sum is often unbounded. There are two different modifications to this sum that are commonly applied in the literature to deal with this.

One of these is known as the *average reward* formulation. Here the goal is to find a policy that maximizes the average reward per unit time. Formally, we maximize:

$$X = \mathop{\mathrm{E}}_{\text{trajectories}}\left[\lim_{\tau_{\max}\to\infty} \frac{1}{\tau_{\max}} \sum_{\tau_i=0}^{\tau_{\max}} r_i^{\tau_i}\right] \tag{2.5}$$

This formulation is intuitively close to the infinite sum above while still remaining bounded. Unfortunately, it can be difficult to work with and so we refer to the interested reader to Puterman (1994) and instead use the next approximation.

The reformulation we use is known as the *discounted reward* formulation. It is called this because future rewards are discounted by a multiplicative factor, $\gamma \in (0, 1)$, for each unit of time that passes. Formally, we maximize:

$$X = \mathop{\mathrm{E}}_{\text{trajectories}}\left[\sum_{i=0}^{\infty} \gamma^{\tau_i} r_i^{\tau_i}\right] \tag{2.6}$$

Here, as above, $r_i^{\tau_i}$ refers to the reward received in step i, which is initiated at time τ_i.

The discount factor, γ, can be justified in a number of ways. It could be viewed as a simple trick to make the sum bounded. It could be viewed as an inflation rate, decreasing the usefulness of future rewards and encouraging the agent to act now rather than later. It can also be viewed as a probability of catastrophic failure stopping the infinite sum. In general we choose $\gamma \approx 0.95$ and use unit time intervals.

2.2 Dynamic Programming Solutions for SMDPs

Markov Decision Processes and methods for solving MDPs have been known since the 1960s (Bellman, 1957). While these methods are only practical for small models, they give a good foundation for our approach. These methods for finding policies also work with Semi-Markov Decision Processes, and we present them here in that framework.

These approaches define functions of the state that reflect the long term desirability of being in a particular state. There are two standard functions that are defined.[5] Each is defined with respect to a particular policy, usually either the current policy, π, or the optimal policy, π^*.

One of these standard functions is known as the *state value function* or just the *value function*, $V : \mathcal{S} \rightarrow \Re$. It returns the expected sum of discounted reward for following the policy from the given state:

[5]Traditionally these functions are represented as tables of values.

$$V^\pi(s) = \mathop{\mathrm{E}}_{\text{trajectories}} \left[\sum_{i_s=0}^{\infty} \gamma^{\tau_{i_s}} r_{i_s}^{\tau_{i_s}} \right] \tag{2.7}$$

$$\begin{aligned} &= R(s, \pi(s)) + \mathop{\mathrm{E}}_{P_{s,\pi(s)}(s',t_1)} \Bigg(\\ &\quad \gamma^{t_1} \Big[R(s', \pi(s')) + \mathop{\mathrm{E}}_{P_{s',\pi(s')}(s'',t_2)} \Big(\\ &\quad \gamma^{t_2} \big[R(s'', \pi(s'')) + \mathop{\mathrm{E}}_{P_{s'',\pi(s'')}(s''',t_3)} (\\ &\qquad \vdots \\ &\qquad)] \Big) \Big] \Bigg) \end{aligned} \tag{2.8}$$

$$= R(s, \pi(s)) + \mathop{\mathrm{E}}_{P_{s,\pi(s)}(s',t)} \left[\gamma^t V^\pi(s') \right] \tag{2.9}$$

where i_s is the ith step from state s, r_{i_s} is the reward for that transition, τ_{i_s} is the time since we first left state s that the step is taken, and $\mathrm{E}_{P(x)}$ is the expectation over the probability distribution P. In particular,

$$\mathop{\mathrm{E}}_{P_{s,a}(s',t)}(X) \equiv \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s', t) X \, \mathrm{d}t \tag{2.10}$$

The other standard function is known as the *state-action value function* or *Q function*, $Q : \mathcal{S} \times \mathcal{A} \to \Re$. It returns, for a particular state and action, the expected sum of discounted reward for being in that state and performing that action once, and henceforth following the policy (Watkins and Dayan, 1992). We can define this formally as:

$$Q^\pi(s, a) = R(s, a) + \mathop{\mathrm{E}}_{P_{s,a}(s',t)} \left[\gamma^t V^\pi(s') \right] \tag{2.11}$$

We can expand the expectations in the above functions to get the following standard recursive forms for the equations:

$$\begin{aligned} Q^{\pi}(s,a) &= R(s,a) \\ &\quad + \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t V^{\pi}(s') \, \mathrm{d}t \end{aligned} \tag{2.12}$$

$$V^{\pi}(s) = Q^{\pi}\left(s, \pi(s)\right) \tag{2.13}$$

Solving these functions for a fixed policy is known as *value determination*. If the Q and V functions are represented as tables of values, then we can replace the equality signs above with assignments to get the following:

$$\begin{aligned} Q^{\pi}(s,a) &\leftarrow R(s,a) \\ &\quad + \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t V^{\pi}(s') \, \mathrm{d}t \end{aligned} \tag{2.14}$$

$$V^{\pi}(s) \leftarrow Q^{\pi}\left(s, \pi(s)\right) \tag{2.15}$$

If these assignments are executed repeatedly for all state action pairs, then the table of values converges to the value of the functions defined in equations 2.12 and 2.13.

We can use the Q and value functions to improve the policy. In particular, if we define the policy π^* as:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s,a) \tag{2.16}$$

then we can solve the set of equations 2.12, 2.13 and 2.16 to obtain the optimal policy: the policy that maximizes the sum of discounted rewards from each state.

Solving this set of simultaneous equations can be done in a number of ways. Again, assuming we have tables of values representing the various functions, we can replace the equalities with assignments:

$$\begin{aligned} Q^{*}(s,a) &\leftarrow R(s,a) \\ &\quad + \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t V^{*}(s') \, \mathrm{d}t \end{aligned} \tag{2.17}$$

$$V^{*}(s) \leftarrow Q^{*}\left(s, \pi(s)\right) \tag{2.18}$$

$$\pi^*(s) \leftarrow \operatorname*{argmax}_{a \in A} Q^*(s,a) \tag{2.19}$$

Again, as long as all points in the domain of all the functions are updated infinitely often, this set of update rules converges in the limit to the optimal policy. There are some named schemes of updating these functions.

Value iteration is a scheme that alternates between updating a state in the Q and value functions, and then updating the policy for that state. A more common style of value iteration does not store the policy. As the action stored in the policy for a particular state is updated after the Q and value functions for that state, the policy is always up-to-date. Hence it is not necessary to store the policy; it can be calculated from the Q function when it is used. A third style of value iteration only stores the value function. The Q values for a particular state are calculated, the value function is updated, and then the Q values are discarded.

At the other extreme, the *policy iteration* scheme updates the Q and value functions for the current policy until they converge without changing the policy. This is a value determination step as described above. Once the values have converged, the policy is updated at each state. These two phases alternate until the policy converges.

Prioritized sweeping (Moore and Atkeson, 1993) (see also Peng and Williams, 1993) is a particularly effective scheme for performing these updates. This scheme is a form of value iteration that controls the order in which states are updated. The states are entered in a priority queue. At each step, the state with the highest priority is updated. The process of updating a state has some extra complexity in order to handle the priorities. First, the value of the state is updated and the change in value, ΔV, is recorded temporarily. Next, the priority of that state is reset to zero. Finally, any state that might be affected by that change in value (any state that has a transition into the updated state) has its priority increased by ΔV. As in other schemes, updating continues until the values converge.

All of the above techniques have been used in practice, although they are all rather slow (polynomial in $|\mathcal{S}|$, which is in turn exponential in the dimensionality of the state space). We refer to them as *traditional* reinforcement learning techniques. We have found that Prioritized Sweeping is generally the most effective and use it in this thesis as both a base-line to compare against, and as a method for solving abstract SMDPs.

2.3 Model Free vs. Model Based Reinforcement Learning

The approaches to finding a policy discussed above fall into a category known as *model based* reinforcement learning because an explicit model of the SMDP is used when finding a policy. It is assumed that the agent has explored the world and learned approximations to P and R. In many algorithms this sampling of P and R occurs in parallel with the calculation of V and Q. This is still considered model based reinforcement learning.

Another approach is the *model free* approach. In this approach the SMDP is never modelled explicitly. Rather the policy is learned directly while exploring the state space. The model free techniques still assume an underlying SMDP, even if it is not modelled explicitly.

The normal model free approach is to convert equations 2.17–2.19 into incremental update functions. The functions are updated when transitions are seen in the world. The new value of the function at a particular location is a weighted average of its old value and the value estimated using the transition the agent has just seen; the weight in this average is known as the *learning rate*. Equations 2.20–2.22 show these formulae, with $\eta \in [0, 1]$ being the learning rate. In these formulae we assume that only Q is being learned and that V and π are calculated from Q when needed. If η is decreased in an appropriate manner as learning progresses, then the table representation of Q converges, with probability 1, to the correct value defined in equation 2.12.

$$\begin{aligned} Q(s,a) &\leftarrow (1-\eta)Q(s,a) + \eta\left(R(s,a) + \gamma^t V(s')\right) && (2.20) \\ V(s) &= \max_a Q(s,a) && (2.21) \\ \pi(s) &= \operatorname*{argmax}_{a \in A} Q(s,a) && (2.22) \end{aligned}$$

There are also intermediate approaches between model free and model based reinforcement learning. One intermediate approach uses a *generative model*. With a generative model we do not have an explicit representation of P, but we can obtain samples from P. This type of model is a simulator rather than a representation of the world. This differs from sampling directly from the world in that the agent can choose which state/action pair it samples from without being forced to follow a trajectory through the state space. Two other intermediate points on the model free/model based spectrum are to allow the agent to reset to either a fixed state, or to a random state, and then sample trajectories from there. The TTree algorithm described in Chapter 5 assumes a generative model has already been learned or supplied. We then concentrate on solving the SMDP given this generative model.

2.4 Improving Problem Solving Speed

All the above techniques suffer from a serious problem: exponential state explosion. The number of states in an SMDP is exponential in the number of variables that describe the world. Exploring and updating all these states is too slow.

A number of techniques have been used to help overcome exponential state explosion and solve large SMDPs. One of these is to use a function approximation technique to store the value function of the current problem. All function approximation techniques have a *generalization bias* allowing them to predict the function values for unseen states or actions. If this generalization bias matches a problem then this technique is effective. If the bias is incorrect for a problem then the technique may be very slow, or even fail to converge to an answer at all.

There are also techniques to abstract the SMDP into a simpler, usually approximate, form. State abstraction refers to the technique of grouping many states together and treating them as one abstract state. Temporal abstraction refers to techniques that group sequences of actions together and treat them as one abstract action. There is a strong relationship between function approximation techniques and these abstraction techniques. Most current function approximation techniques have a very similar effect to state abstraction.

2.4.1 Styles of state abstraction

While in general *state abstraction* refers to the technique of grouping many states together and treating them as one *abstract state*, this has been done in the literature in a number of different ways. As already noted, there is a strong relationship between state abstraction and function approximation, and many of the techniques discussed below have elements of both. We are including in this section some forms of function approximation that have effects similar to state abstraction.

The simplest approach to state abstraction is to manually make a new ‘abstract’ SMDP. Each state in the abstract SMDP corresponds to a group of states in the original SMDP. This new SMDP can then be solved using a traditional technique.

Another approach to state abstraction is to use a CMAC function approximator (Albus, 1981; Miller et al., 1990; Lin and Kim, 1991; Sutton, 1996) to represent the Q or value functions. The CMAC represtation uses a user supplied division of the state space into overlapping regions. Each of these regions is assigned a value in the CMAC. The output of the CMAC at a particular state is the average of the values of the regions that contain that state. This style of function approximation is easily differentiable and allows incremental update.

A third approach to state abstraction is to use a linearly interpolating function approximator for the value function (Gordon, 1995). In this approach, the user breaks the state space into a series of regions defined by points on their border. Each of these points is given a value. The value of the function approximator within a region is a linear combination of the values at the points defining that region.

All of the previous approaches have required the user to pre-specify how a state space is discretized. We now discuss some styles of state abstraction that attempt to discover how the state should be abstracted. These methods of forming a state abstraction are sometimes described as discretizing a state space, and this description is used even when the state space is already discrete. In this case it may help the reader to think of discretization as re-discretization; the original discretization is set aside and then useful parts of it are re-introduced.

The first group of automatic state abstraction techniques are known as *variable resolution* techniques (Moore, 1994; Munos and Moore, 1999). In these techniques the world is initially discretized into a coarse grid. The algorithms then iterate trying to detect regions where the resolution is too low and increasing the resolution in those regions. The technique is known as variable resolution because the resolution can be increased in some regions and not others. Normally, the discretization is represented as a tree structure. Each grid region is a leaf in the tree. The resolution is increased in a region by replacing that region's leaf with an internal node and more leaves. The branching factor is usually either two, or 2^d where d is the dimensionality of the space. A region is divided either halfway along its longest axis or halfway along each axis.

The second group of automatic state abstraction techniques are known as *regression tree* techniques (Breiman et al., 1984; Quinlan, 1992; Chapman and Kaelbling, 1991; McCallum, 1995; Uther and Veloso, 1998). Like variable resolution techniques, regression tree techniques also discretize the state space using a tree structure, and also increase the resolution of that discretization by growing the tree in some regions of the state space. The difference between the two types of algorithms is in the method by which the tree is grown.

Variable resolution techniques make a binary distinction at each leaf in the tree, “Should this node have its resolution increased?” Regression tree techniques ask the additional question, “How should I increase the resolution of this node?” In particular, regression tree techniques attempt to find the right attribute and location to split a node, rather than just dividing down the middle. This thesis uses regression tree techniques.

2.4.2 Styles of temporal abstraction

Temporal abstraction refers to techniques that group sequences of actions together and treat them as one *abstract action*. There are two main styles of temporal abstraction described in the literature. The first style (*e.g.* Dietterich, 1998; Parr and Russell, 1998) has a strict, pre-defined hierarchy of actions. Each action can use only the actions in the next lower level of the hierarchy to achieve its goal. Often this hierarchy includes some state abstraction related to the temporal abstraction (Dietterich, 2000). Hengst (2000) has recently generated hierarchies of a limited form by finding a fixed ordering of the state variables. While this approach is effective, it is limited by requiring the same variable ordering in all regions of the state space.

The second main style (Sutton et al., 1998; Precup, 2000) revolves around defining policies with termination criteria, known as *options*. If only the termination criteria are defined then the policies can be learnt, but the termination criteria are required. These policy/end point pairs can then be used as base level actions in an SMDP. Once an option is selected to be executed, its policy is followed until the termination criteria are fulfilled. The combined series of base level actions is treated as one, temporally extended, action by the SMDP. McGovern and Barto (2001) have recently managed to automatically find options for the class of endpoints defined by ‘bottlenecks’ in the state space.

This thesis introduces a new style of temporal abstraction similar to options (Sutton et al., 1998), but we remove the requirement for defining the end points of the option. Our abstract actions are simply policies. It is up to the algorithm using these abstract actions to decide in which regions of the state space each abstract action is followed. In fact, arbitrary controllers can be plugged in to the algorithm, but we do not consider non-markovian controllers in the proof of correctness or experiments.

Consider, for example, a robot that walks through a maze as in Chapter 1. This robot is legged, and the walking motion is non-trivial. One might imagine a set of four policies, each of which walks the robot in one the four cardinal directions, north, south, east and west. Each of these policies depends only upon the state variables that define the leg position, and not upon the state variables defining the position of the robot in the maze. An algorithm solving the walking robot in a maze problem need only learn which policy to use in which region of the maze.

2.4.3 Subroutines vs. polling

Another dimension along which we can classify temporal abstraction algorithms is the subroutines vs. polling dimension. The first class of algorithms treats abstract actions as *subroutines* – once an abstract action is selected it executes until completion. The second class relies upon *polling* – the abstract controller is called every time a decision is needed, it selects which abstract action to use for that decision, and then the abstract action selects a base action to perform.

There are advantages and disadvantages to each of these approaches. The reward structure of a subroutine is defined by its termination conditions and hence subroutines have the advantage that they can be optimized independently of the reward structure of the rest of the problem. It is even possible to optimize a number of subroutines in parallel. The disadvantage of subroutines is that they limit the policies that can be represented using just the abstract actions – it is not possible to use only the first half of a subroutine.

Polling algorithms are not as limited in their optimality. As long as in every state, each base level action is used by at least one of the abstract actions, a polling algorithm can represent any policy. However, it is much harder to optimize the abstract actions in a polling algorithm independently of the problem they are helping to solve.

Many previous reinforcement learning algorithms for temporal abstraction, and most of the algorithms with theoretical guarantees, fall in the subroutine category. We investigate the polling approach. Rather than attempt to learn abstract actions and optimize them for the current problem, we extract abstract actions from previous problems.

2.5 Decision and Regression Trees

There are numerous functions defined in the preceding sections of this chapter, for example the Q and value functions for an SMDP. Throughout this thesis we use trees to represent functions of various types. This section is a brief introduction to tree-based representation of functions.

A function maps an input space, the *domain*, onto an output space, the *range*. In a tree based representation, nodes in the tree correspond to regions in the domain. An internal node contains information that divides the region amongst its children. A leaf node contains an element of the range of the function. The function represented by the tree maps the region of the domain corresponding to a leaf into the element of the range stored in that leaf node.

There are a number of different types of trees depending upon whether the domain and/or range are discrete or continuous. As noted above when discussing decision and regression trees, there are also different types of trees which vary on how internal nodes divide their region of space. We make a distinction between variable resolution trees (Friedman et al., 1977) and regression or decision trees (Breiman et al., 1984; Quinlan, 1992).

In summary, a *variable resolution tree* has internal nodes that divide their region of state space either down the middle of the region's longest dimension, or down the middle of each dimension. When growing this type of tree, the learner simply needs to decide whether a leaf node should have increased resolution. The function to be represented has no effect on *how* the resolution is increased. This is in contrast to a *regression tree* or *decision tree*. In this type of tree an internal node contains more specific information about how the current region is divided. For example, it is most common for an internal node to divide along only one dimension. If that dimension is continuous then the internal node stores where along that dimension the region is divided. Other ways of dividing the region have also been investigated, for example linear divisions of the region are not uncommon (Utgoff and Brodley, 1991; Murthy et al., 1994). For reasons of simplicity we stick with axis-parallel divisions in this thesis.

In this thesis we concentrate on the regression tree representation. The important distinction is that the internal nodes in a regression tree store more information about the function being represented. This is important for our analysis of these functions in Chapter 4, and for accurately building an abstract SMDP in Chapter 5.

2.6 Domains

To experimentally test our algorithms we have simulated a number of different domains. In this section we describe those domains. We should note that not all these domains were used with all the algorithms. Additionally, different algorithms have different ability to handle ordered discrete attributes and so we may have multiple representations for a domain even if the underlying dynamics are the same.

2.6.1 A heated corridor domain

Our first domain is a robot 2 dimensional domain with a robot moving down a heated corridor. The two state dimensions are location and temperature. Location is an ordered, discrete attribute and the agent needs to distinguish between all locations to represent the correct value function. Temperature is continuous, but can be divided into three qualitatively different regions - cold, normal and hot. The agent is not given this qualitative division.

In this domain the length of the corridor is divided into three sections: sections A, B and C. These three sections of corridor each have a different base temperature. A is cooled, B is normal temperature and C is heated. The robot is temperature sensitive, but also has limited temperature control on board. The robot starts at a random position in the corridor and is rewarded when it reaches the right-hand end of the corridor. The robot must learn to move towards the correct end of the corridor, to turn its heater on in section A and to turn its cooler on in section C.

The robot's actions make it move and change its temperature control settings. The robot has 6 actions it can perform: move forward or backward 1 unit, each with either heater on, cooler on, or neither on. The robot moves nondeterministically as a function of its temperature. While the robot's temperature is in the $(30^\circ, 70^\circ)$ range, it successfully moves with 90% probability, otherwise it only moves with a 10% probability.

The robot's temperature is the base temperature of that section of corridor adjusted by the robot's temperature control unit. In our experiments, the corridor was 10 units long. Sections A , B and C were 3, 4 and 3 units in length respectively, with base temperature ranges of $(5^\circ, 35^\circ)$, $(25^\circ, 75^\circ)$ and $(65^\circ, 95^\circ)$ respectively. The robot's temperature control unit changed the temperature of the robot to differ from the corridor section base temperature by $(+25^\circ, +45^\circ)$ if the heater was on, and by $(-25^\circ, -45^\circ)$ when the cooler was on. All temperatures and temperature adjustments were independently sampled at each time-step from a uniform distribution over the range given. Figure 2.1 illustrates some of the state transitions for this domain.

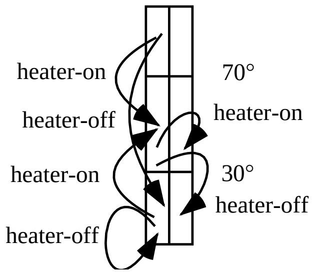


Figure 2.1: Sample transitions for part of the heated corridor domain. The y axis is the robot's temperature. The x axis represents two consecutive locations in the cool section of the corridor. High probability transitions moving right are shown.

2.6.2 A hexagonal soccer domain

Our second domain is a hexagonal grid based soccer simulation called *Hexcer*. Hexcer is similar to, but larger than, the game framework used by Littman (1994) to test Minimax Q Learning. Hexcer consists of a field with a hexagonal grid of 53 cells, two players and a ball (see Figure 2.2). Each player can be in any cell not already occupied by the other player. The ball is either in the center of the board, or is controlled by (in the same cell as) one of the players. This gives a total of 8268 distinct states in the game.

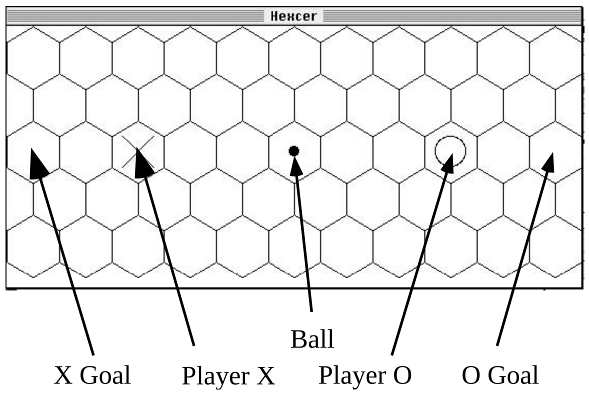


Figure 2.2: The Hexcer board

The two players start in fixed positions on the board, as shown in Figure 2.2. The game then proceeds in rounds. During each round the players make simultaneous moves to a neighboring cell in any of the six possible directions. Players must specify a direction in which to move, but if a player attempts to move off the edge of the grid, it remains in the same cell. Once one player moves onto the ball, the ball stays with that player until stolen by the other player.

When the two players move so that they would end up in the same cell, one of them succeeds and the other fails, remaining in its original position. The choice of who moves into the contested cell is usually made randomly with each player having an equal chance. The single exception is when one of the players is already in the contested cell (for example, if it tried to move into a wall and so did not move). In this case the player already occupying the cell is considered to win and remains in that cell. The ball, if it was in the control of one of the players, moves into the contested cell regardless of which player wins. This allows the ball to change hands.

Players score by taking the ball into their opponent's goal. When the ball arrives in a goal the game ends. The player guarding the goal loses the game and gets a negative reward. The opponent receives an equal magnitude, but positive, reward. It is possible to score an own goal.

In our Hexcer experiments, each state is represented by a vector of 7 attributes. These attributes are the x and y coordinates of each player, the location of the ball (On X, on O or in the middle of the board), and the difference in x and y coordinates between the two players. This representation is redundant in that the difference in x and y coordinates between the two players can be calculated from the other attributes.

The simple form of decision used by both Continuous U Tree and TTree in their state trees can only divide the space parallel to the axes of the input space. Other more complex types of decision exist in the regression tree literature (*e.g.* Utgoff and Brodley, 1991), but simple decisions have been shown to be remarkably effective , and much faster to find than more complex decisions. By adding redundant attributes that are linear combinations of the primary attributes we can allow the algorithm to find diagonal splits.

The hexcer domain is different from other domains described in this section in that it has multiple players. If one of the players has a fixed, although possibly stochastic, policy then this domain still fits neatly in the SMDP framework. The fixed player is simply part of the domain as viewed by the other player. When both players are learning, then the SMDP, as viewed by each player, becomes non-stationary. We do not cover this in detail in this thesis but refer the reader to the literature on stochastic games (*e.g.* Bowling and Veloso, 2000).

2.6.3 The towers of hanoi domain

The Towers of Hanoi domain is not a single domain, but rather a series of domains of different sizes. Each of these domains consists of 3 pegs, $\{P_0, P_1, P_2\}$, on which sit N disks, $\{D_1, D_2, \ldots, D_N\}$. The number of disks determines the size of the domain. Each disk is of a different size, D_1 being the smallest, D_2 the next smallest, *etc.*, and they stack such that smaller disks always sit above larger disks. Some example states from the eight disc problem are shown in Table 2.1. There are six actions, shown in Table 2.2, which move the top disk on one peg to the top of one of the other pegs. An illegal action, trying to move a larger peg on top of a smaller peg, results in no change in the world. The object is to move all the disks to a specified peg, usually peg P_2; a reward of 100 is received in this state. All actions take one time step. The decomposed representation we used has a boolean variable for each disk/peg pair. These variables are true if the disk is on the peg.

Example	Disks on peg		
state ID	P_0	P_1	P_2
s_1	D_1		$D_2D_3D_4D_5D_6D_7D_8$
s_2	$D_1D_4D_7$	D_2D_5	$D_3D_6D_8$
s_3	$D_1D_4D_7$	$D_2D_5D_8$	D_3D_6
s_4	$D_1D_4D_8$	$D_2D_5D_7$	D_3D_6

Table 2.1: A set of sample states in the Towers of Hanoi domain.

Action	Move Disc	
	From Peg	To Peg
a_0	P_0	P_1
a_1	P_0	P_2
a_2	P_1	P_2
a_3	P_1	P_0
a_4	P_2	P_1
a_5	P_2	P_0

Table 2.2: The base level actions in the Towers of Hanoi domain

2.6.4 The walking robot domain

This domain has already been introduced in Chapter 1 with Robbie the robot, but we cover it more formally here. This domain simulates a two legged robot walking through a maze. The two legs are designated left and right. With a few restrictions, each of these legs can be raised and lowered one unit, and the raised foot can be moved one unit in each of the four compass directions: north, south, east and west. The legs are restricted in movement so that they are not both in the air at the same time. They are also restricted to not be diagonally separated, *e.g.* the right leg can be either east or north of the left leg, but it cannot be both east and north of the left leg.

More formally, we represent the position of the robot using the two dimensional coordinates of the right foot, $\langle x, y\rangle$. We then represent the pose of the robot's legs by storing the three dimensional position of the left foot relative to the right foot, $\langle \Delta x, \Delta y, \Delta z\rangle$. We represent East on the $+x$ axis, North on the $+y$ axis and up on the $+z$ axis. The formal restrictions on movement are that Δx and Δy cannot both be non-zero at the same time and that each of Δx, Δy and Δz are in the set $\{-1, 0, 1\}$. A subset of the state space is shown diagrammatically in Figures 2.3 and 2.4. These figures are too small to show the entire global state space. They also ignore the existence of walls.

The robot walks through a grid with a maze imposed on it. Some example 10×10 mazes are shown are shown in Figure 2.5. The mazes have the effect of blocking some of the available actions: any action that would result in the robot having its feet on either side of a maze wall fails. Any action that would result in an illegal leg configuration fails and gives the robot reward of -1. Upon reaching the grey square in the maze the robot receives a reward of 100.

In some of the algorithms in this thesis we do not handle ordered discrete attributes such as the global maze coordinates, x and y. In these cases we transform each of the ordered discrete attributes into a set of binary attributes. There is one binary attribute for each ordered discrete attribute/value pair describing if the attribute is less than the value. For example, we replace the x attribute with a series of binary attributes of the form: $\{x < 1, x < 2, \ldots, x < 9\}$. The y attribute is transformed similarly. The Continuous U Tree algorithm (see Chapter 3) can handle ordered discrete attributes because it applies a similar transformation automatically.

In addition to the mazes above, we use the 'gridworlds' shown in Figures 2.6 and 2.7 for experiments. It should be remembered that the agent has to walk through these grids.

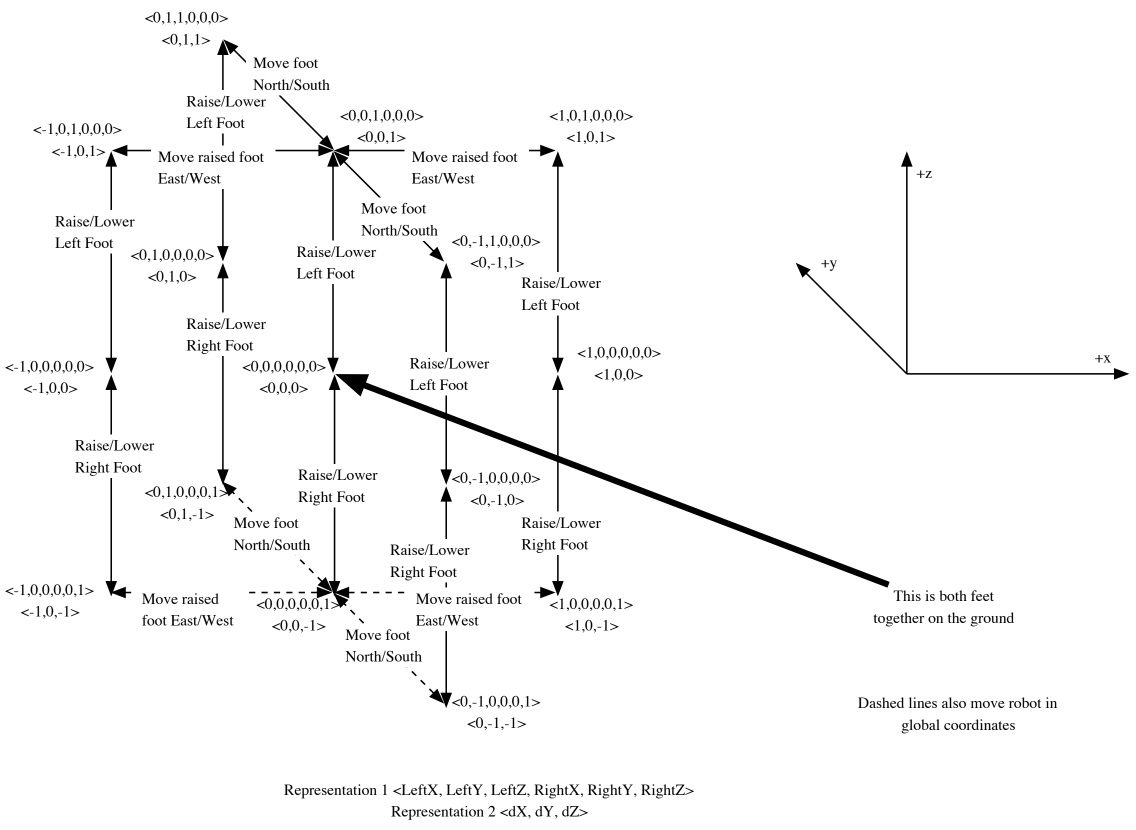


Figure 2.3: The local transition diagram for the walking robot domain without walls. This shows the positions of the feet relative to each other. Solid arrows represent transitions possible without a change in global location. Dashed arrows represent transitions possible with a change in global location. The different states are shown in two different coordinate systems. The top coordinate system shows the positions of each foot relative to the ground at the global position of the robot. The bottom coordinate system shows the position of the left foot relative to the right foot.

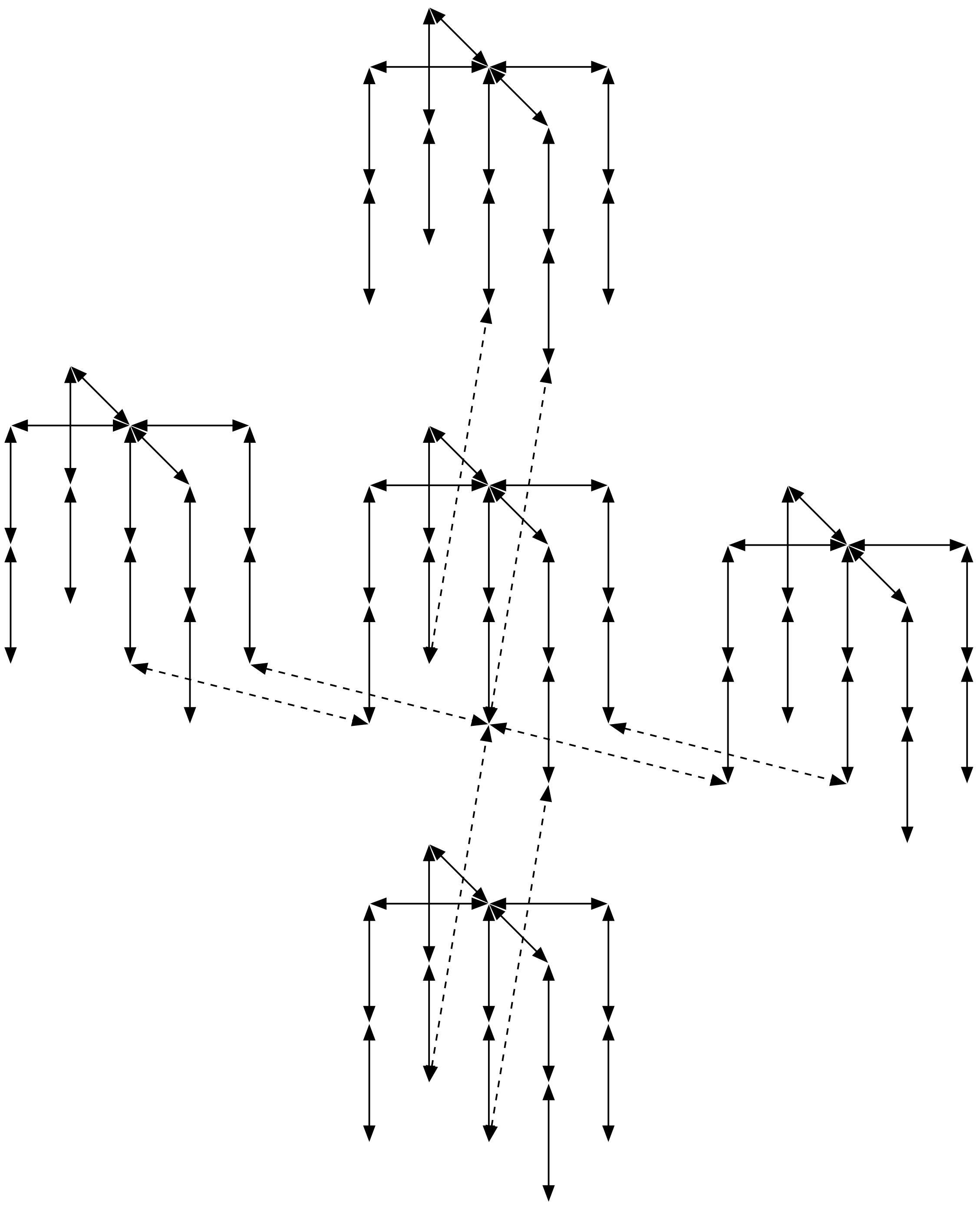


Figure 2.4: A subset of the global transition diagram for the walking robot domain. Each of the sets of solid lines is a copy of the local transition diagram shown in Figure 2.3. As in that figure, solid arrows represent transitions that do not change global location and dashed arrows represent transitions that do change global location.

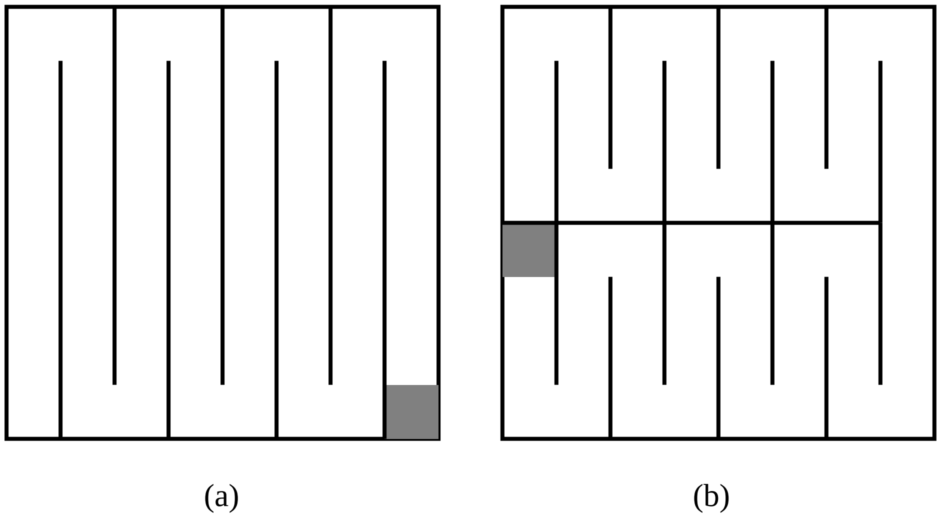


Figure 2.5: A pair of example mazes for our robot. The robots's goal in each maze is to reach the grey patch. These are the same mazes as Figure 1.1, repeated for ease of reference.

Unless stated otherwise in the experiments we have a reward of 100 in the bottom right square of the gridworld.

2.6.5 The taxi domain

The taxi domain (Dietterich, 1998) is a gridworld that has been used by other researchers to evaluate their temporal abstraction algorithms. The idea behind the domain is that there is a taxi moving about a gridworld. There are four taxi stands in the world. The taxi must go to a taxi stand, pick up a passenger, take the passenger to the destination taxi stand and then drop the passenger off. The gridworld is shown in Figure 2.8. The four squares marked with the letters R, G, B and Y are the four taxi stands.

The state has four dimensions: the x and y location of the taxi, the location of the passenger (either one of the four taxi stands, or in the taxi) and the destination of the passenger (one of the four taxi stands). There are six actions the agent can perform: it can move the taxi in one of the four cardinal directions, it can also pick up and put down the passenger. There is a reward of -10 for executing the pickup and putdown actions when not in the correct location. There is a reward of 100 for getting the passenger to their destination. This formulation is slightly different from that of Dietterich. He uses a final reward of 20 and a cost of -1 for each step. The reasons for this change are discussed in

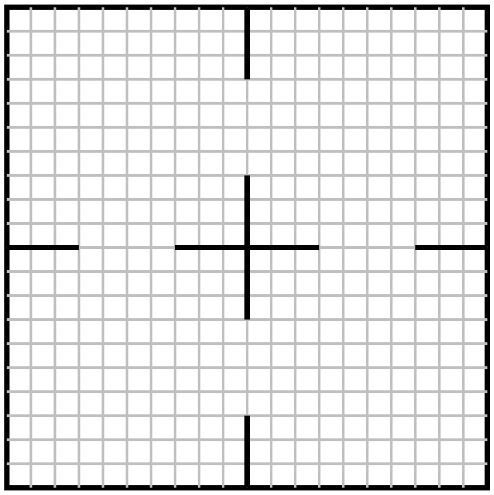


Figure 2.6: A set of four 10×10 rooms for our robot to walk through.

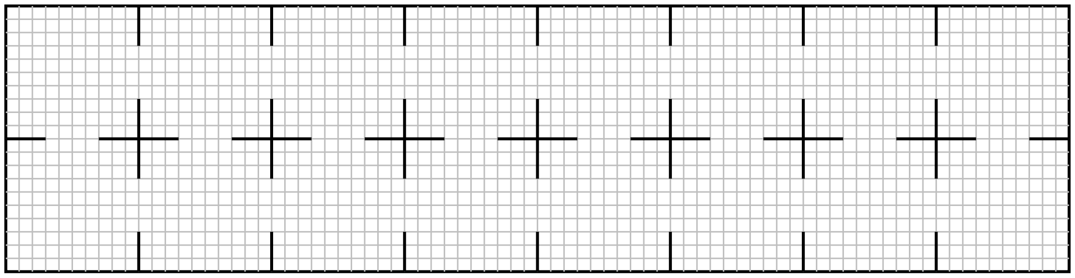


Figure 2.7: A set of sixteen 10×10 rooms for our robot to walk through.

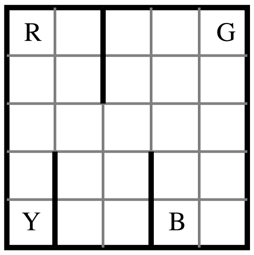


Figure 2.8: The simple gridworld of the Taxi domain.

Chapter 5.

Chapter 3

Continuous U Tree

In the introduction and technical overview chapters we introduced the SMDP framework and also noted that it has difficulty scaling to large problems. In this chapter we describe our Continuous U Tree algorithm; a state abstraction algorithm extending the work of Chapman and Kaelbling (1991) and McCallum (1995). This chapter shows how regression trees can be used for state abstraction reinforcement learning. It also introduces concepts about regression tree based reinforcement learning that we build on in later chapters.

Continuous U Tree is a state abstraction algorithm. In many SMDPs there are parts of the state space where the exact position in state space is irrelevant to the agent. All the states in such a region are equivalent and can be replaced by a single state, reducing the size of the state space and hence making the problem easier to solve.

The G algorithm (Chapman and Kaelbling, 1991) and the U Tree algorithm (McCallum, 1995) are two similar algorithms that perform state abstraction by re-discretizing propositional state spaces. A policy can then be found for the new discretization using traditional techniques. Both algorithms start with the world as a single abstract state and recursively split it where necessary. The Continuous U Tree algorithm extends these algorithms to work with continuous state spaces rather than propositional state spaces.

To evaluate Continuous U Tree we used two domains, a small two dimensional domain and a larger seven dimensional domain. The small domain, a robot travelling down a corridor, shows some of the main features of the algorithm. The larger domain, hexagonal grid soccer, is similar to the one used by Littman (1994) to investigate Markov games. It is ordered discrete rather than strictly continuous. We increased the size of this domain, both in number of states and number of actions per state, to test the generalization capabilities of our algorithm.

3.1 Markov Decision Problems

In the previous chapter we introduced Semi-Markov decision problems as a generalization of Markov Decision Problems. This algorithm was developed in the MDP framework and, while the algorithm is readily extendable to the SMDP framework, we present the algorithm in its original form here. An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$., just like an SMDP. The difference is in the definition of P. In an MDP, $P_{(s,a)}(s')$ is a probability distribution over states, not states and times.

We can then state the MDP form of the Bellman equations, defining the Q and value functions:

$$Q(s,a) = R(s,a) + \sum_{s'} P_{(s,a)}(s') \gamma V(s') \tag{3.1}$$

$$V(s) = \max_a Q(s,a) \tag{3.2}$$

3.2 Continuous U Tree

The G-Algorithm (Chapman and Kaelbling, 1991) and U Tree (McCallum, 1995) both apply propositional decision tree techniques to reinforcement learning. Continuous U Tree is an extension to these techniques that is capable of directly handling both propositional and continuous-valued domains. Continuous U Tree, uses decision tree learning techniques (Breiman et al., 1984; Quinlan, 1986) to find a discretization of the continuous state space.

Continuous U Tree is different from traditional reinforcement learning algorithms in that it does not require a prior discretization of the world into separate states. The algorithm takes a state space decomposed into a series of continuous, ordered discrete or discrete attributes, and automatically discretizes it to form a discrete MDP. Any discrete, state-based, reinforcement learning algorithm can then be used on this MDP.

In both U Tree and Continuous U Tree there are two distinct concepts of state. In Continuous U Tree the first type of state is the fully continuous state of the world that the agent is moving through. We term this *sensory input*, or base level state. The second type of state is the position in the discretization being formed by the algorithm. We use the term abstract state to describe this second type of state. Each abstract state is an area in the sensory input space, and each sensory input falls within an abstract state. In many previous algorithms these two concepts were identical. Sensory input was discrete and each different sensory input corresponded to a state.

The translation between these two concepts of state is in terms of a *state tree* that describes the discretization. Each node in this binary tree corresponds to an area of sensory input space. Each internal node has a *decision* attached to it that describes a way to divide its area of sensor space in two. These decisions are described in terms of an attribute of the sensory input to split on, and a value of that attribute. Any sensory input where the relevant attribute falls above or below the stored value is passed down the left or right side of the tree respectively. Leaf nodes in the tree correspond to abstract states. A state tree enables generalization – abstract states correspond to regions in sensory input space.

We refer to each step the agent takes in the world as a *transition*. Each saved transition is a tuple of the starting sensory input, I, the action performed, a, the resulting sensory input, I' and the reward obtained for that transition, r: $\langle I, a, I', r\rangle$. The sensory input is itself a vector of values. These values can be continuous.

As the agent is forming its own discretization, there is no prior discretization that can be used to aggregate the data. (If a partial prior discretization exists, this can be utilized by the algorithm by initializing the state tree with that discretization, but this is not required.) Only a subset of the transitions need to be saved, but they need to be saved with full sensor accuracy.

Each transition $\langle I, a, I', r\rangle$ is used to generate one *datapoint* $\langle I, a, q(I, a)\rangle$ – a triple of the sensory input, I, the action, a, and a value, $q(I, a)$. The value of a datapoint is its expected reward for performing that transition and behaving optimally from then on. The calculation of this value is discussed below.

Continuous U Tree forms a discretization by recursively growing the state tree. The algorithm, shown in Table 1, is as follows: Initially the world is considered to be a single abstract state with an expected reward, $V(s) = 0$. The state tree is a single leaf node corresponding to the entire sensory input space. The algorithm then loops through a two phase process: Datapoints are accumulated for learning in a data gathering phase, and then a processing phase updates the discretization. During the data gathering phase the algorithm behaves as a standard reinforcement learning algorithm, with the added step of using the state tree to translate sensory input to a state. It also remembers the transitions it sees. During the processing phase, the values of all the datapoints, $q(I, a)$, are re-calculated. If a significant difference in the distribution of datapoint values is found within an abstract state, then that abstract state is split in two. A split adds two new leaves to the state tree representing the two new abstract states formed by dividing the old abstract state. The MDP defined over the new discretization is solved to find the Q function over the abstract states which can in turn be used to find a policy. Initializing the abstract state values to the values in the previous discretization and using an incremental algorithm makes updating these values relatively fast. Deciding when to execute a pass through the processing phase is a parameter to the algorithm. In our experiments we perform a pass of the processing phase at the end of every trial. Trials end when an absorbing state is reached or when 1000 steps have been taken in the world.

This is similar to the recursive partitioning of a decision or regression tree, except that the values of the datapoints can change when more data is gathered and the MDP is re-solved after each processing phase. Splitting is also breadth-first rather than depth-first. By this we mean that we do not use the same left-recursive algorithm to grow the trees that is used in many supervised decision tree algorithms. Rather, all leaves that need increased resolution are expanded at the same time, and then the abstract MDP is re-solved before testing the leaves again.

Procedure 1: The Continuous U Tree algorithm

```
loop
  while Gathering Data do
    Gather Data (see Procedure 2)
  end while
  Expand Tree (see Procedure 3)
end loop
```

The value of a datapoint $q(I, a)$ is calculated using a modification of the Bellman equations. Using the value for the resulting state of a transition $V(s')$ and the recorded reward for the transition r, we can assign a value to the corresponding datapoint:

$$q(I, a) = r + \gamma V(s') \tag{3.3}$$

Having calculated these datapoint values, Continuous U Tree then tests if the statistical

Procedure 2: Gather Data

```
$I \leftarrow$ current sensor vector
$s \leftarrow GetLeaf(I)$ (see Procedure 4)
$a \leftarrow \text{argmax}_{a'} Q(s, a')$
Perform action $a$
$I' \leftarrow$ current sensor vector
$r \leftarrow$ reward for previous transition
$D \leftarrow D \cup \{\langle I, a, I', r\rangle\}$
$T \leftarrow T \cup \{\langle s, a, GetLeaf(I'), r\rangle\}$
UpdateQFunction($T$)
```

distribution of the values varies systematically within any state, and, if so, finds the best single decision to use to divide that state in two. This is done using decision tree techniques. Each attribute in the sensory input is considered in turn. The datapoints are sorted according to that attribute. The algorithm loops through this sorted list and a trial split is added between each consecutive pair of datapoints. This split divides the datapoints into two sets. These two sets are compared using a *splitting criterion* (see below) which returns a single number describing how different the two distributions are. The trial split that leads to the largest difference between the two distributions is remembered. The “best” split is then evaluated by a fixed *stopping criterion* (see below) to check whether the difference is significant.

Having found a discretization, the task of finding a policy has been reduced to a standard, discrete, reinforcement learning problem: finding Q values for the new abstract states, $Q(s, a)$. This is done by calculating state transition probabilities, $P_{(s,a)}(s')$, and expected rewards, $R(s, a)$, from the recorded transitions and then solving the MDP so specified. We have used both Prioritized Sweeping (Moore and Atkeson, 1993) and conjugate gradient descent on the sum-squared Bellman residuals (Baird, 1995) to solve the MDP defined over this new discretization.

3.2.1 Splitting Criteria: Testing for a difference between data distributions

We tried two different splitting criteria. The first is a non-parametric statistical test – the Kolmogorov-Smirnov test. The second is based on sum-squared error.

Procedure 3: Grow The Tree

```
$D_q \leftarrow \emptyset$
**for all** leaves $l$ **do**
  **for all** points $\langle I, a, I', r\rangle \in D$ such that $GetLeaf(I) = l$ **do**
    $D_q \leftarrow D_q \cup \{\langle I, a, (r + \gamma V(GetLeaf(I'))\rangle\}$
  **end for**
  $K_{\text{best}} \leftarrow \emptyset$
  $P_{\text{best}} \leftarrow 0$
  **for all** possible divisions, $K$, of $l$ **do**
    $P \leftarrow$ evaluation of $K$ using the splitting criterion and $D_q$
    **if** $P > P_{\text{best}}$ **then**
      $P_{\text{best}} \leftarrow P$
      $K_{\text{best}} \leftarrow K$
    **end if**
  **end for**
  **if** $K_{\text{best}} \neq \emptyset$ **then**
    Replace $l$ with $K_{\text{best}}$
  **end if**
**end for**
$T \leftarrow \emptyset$
**for all** $\langle I, a, I', r\rangle \in D$ **do**
  $T \leftarrow T \cup \{\langle GetLeaf(I), a, GetLeaf(I'), r\rangle\}$
**end for**
UpdateQFunction($T$)
```

Procedure 4: Get the leaf for a sensor vector

```
Let node $n \leftarrow root$
**while** $n$ is not a leaf **do**
  $K \leftarrow$ decision criterion in internal node $n$
  **if** $K(I)$ **then**
    $n \leftarrow$ left child of $n$
  **else**
    $n \leftarrow$ right child of $n$
  **end if**
**end while**
return $n$
```

The first splitting criterion is that used by McCallum (1995) in the original U Tree algorithm, which looked for violations of the Markov assumption. If the distribution of datapoint values in each of the two new states was different, then there is a possible violation of the Markov assumption – more information about the world is available by knowing where the agent is within the old abstract state. Knowing how you got into that abstract state might tell you something about where you are, and hence history has become important – a violation of the Markov assumption. The splitting criterion was then a test of the statistical similarity of the distributions of the datapoints on either side of the split. We used the Kolmogorov-Smirnov non-parametric test. This test is based on the difference in the cumulative distributions of the two datasets.

Figure 3.1 illustrates the cumulative distributions of two datasets. The arrow between the two lines marks the maximum difference D between the cumulative probability distributions, C_1 and C_2. This distance has a distribution that can be approximately calculated for two independent, but identically distributed, sets of datapoints regardless of what distribution the sets are drawn from. Assuming the cumulative distributions are C_1 and C_2, with dataset sizes of N_1 and N_2 respectively, the probability that the observed difference could be generated by noise is calculated using the equations in Table 3.1 (Press et al., 1992). The smaller this probability, the more evidence there is of a true difference. If it is small enough to pass the stopping criterion, then a split is introduced.

We investigated a second squared-error based splitting criterion (Breiman et al., 1984). The aim is to approximate the Q values of the transitions, $q(I, a)$, using only the Q values

$$D = \max_{-\infty<x<\infty} |C_1(x) - C_2(x)| \tag{3.4}$$

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \tag{3.5}$$

$$\lambda = \left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right] D \tag{3.6}$$

$$P_{Noise} = 2\sum_{j=1}^{\infty}(-1)^{j-1}e^{-2j^2\lambda^2} \tag{3.7}$$

Table 3.1: The Kolmogorov-Smirnov test equations

of the states, $Q(s,a)$. The error measure for this fit is the sum-squared error.[1]

One good splitting criterion would be to introduce the split, solve the MDP formed over the abstract states, and then see if the two new abstract states formed have different values. The difference between the values of the two new abstract states is a measure of how useful the split is. This is too expensive.

If we make the simplifying assumptions that the Q values of other abstract states do not change, and that there are no transitions between the prospective abstract states, then the Q values of the two prospective abstract states equal the mean of the datapoints in those states. A sum squared error splitting criterion is measuring the weighted sum of the variances of the Q values of the transitions on either side of the split. A second justification for this splitting criterion is the finding of Williams and Baird (1993) that reducing the Bellman residual of a value function increases performance.

Both these splitting criteria can be made more sensitive by performing the tests separately for each action and then combining the results. For the mean-squared error test a weighted sum was used to combine the separate tests. For the Kolmogorov-Smirnov test we multiply the probabilities calculated.

Importantly from an efficiency viewpoint, both of these tests can be done at least partly incrementally. For the sum-squared error, we can keep track of $\sum_n x$ and $\sum_n x^2$ and use these to estimate the variance, giving a resulting time complexity for a series of tests of

[1]This measure is closely related to the t-test used by Chapman and Kaelbling (1991).

$O(n)$.

$$\text{Variance} = \frac{\sum_n x^2 - \frac{\left(\sum_n x\right)^2}{n}}{n-1} \tag{3.8}$$

For the Kolmogorov-Smirnov test, the datasets need to be sorted to find the cumulative probability distributions. This sorting can be performed once for a series of tests by trading $O(n)$ space, but we still need to loop over the distributions to find D for each test making the total time $O(n^2)$.

3.2.2 Stopping Criteria: Should we split?

When learning a regression tree from data, the standard technique is to grow the tree by recursively splitting the data until a stopping criterion is met. Obviously, if all the datapoints in a leaf have the same value, then there is no need to grow the tree. Additionally, there is no effective way to grow the tree if there is no way to separate the data – for example all datapoints fall at the same point in the sensory input space.

More generally, the algorithm should stop when it cannot detect a significant difference between the datasets on either side of the best split in a leaf. Here the word significant can have different meanings. For the Kolmogorov-Smirnov test we use significant to mean statistically significant at the $P = 0.05$ level.

The stopping criterion for the squared-error based test is the reduction in mean squared-error. If the difference between the variance of the entire dataset and the weighted mean variance of the datasets induced by the split is below a threshold, then there is not a significant difference. This test is less theoretically based than the other test, but with careful adjustment of the threshold it produced reasonable results.

In the tree based learning literature, it is well known that stopping criteria often have to be weak to find good splits hidden low in the tree. To stop overly large trees being produced, the trees are often pruned before being used. We do not prune. We assume that little error is introduced by over-discretizing.

3.2.3 Datapoint sampling: What do we save?

In the description of the algorithm above, we noted that transitions are saved. Notably, Continuous U Tree does not need to save all of the transitions. Its generalization capabilities allow it to learn from non-contiguous trajectories through state space. Recording all of the transitions is expensive, both in terms of memory to record the data and in processing time when testing for splits. However, if fewer transitions are recorded then it is harder to detect differences within a leaf.

When limiting the number of saved datapoints, we recorded a fixed number of datapoints per abstract state/leaf in the tree. Even once the number of transitions to store is set, it is still unclear which transitions to store. In the experiments reported we remembered all transitions until the preset per-leaf limit was reached. We then made room for new transitions by discarding a random transition from that leaf. This has the advantage that no bias, which might mislead our splitting or stopping criterion, is introduced.

3.3 Experimental Results

We performed experiments with this algorithm in two domains; the heated corridor domain described in Section 2.6.1, and the hexagonal soccer domain described in Section 2.6.2. As a quick reminder, the heated corridor domain has a robot moving down a corridor. The location in the corridor is 1 dimensional with 10 ordered-discrete states. The corridor has a second dimension, temperature. This dimension is continuous with three qualitatively different ranges of temperature. The hexagonal soccer domain has two agents moving around a hexagonal gridworld, each trying to move a ball to their own goal.

Figure 3.2 shows the discretization and policy for the corridor domain after 3000 steps in the world. The y axis is the temperature of the robot on a scale of $0 - 100$. The x axis is the location of the robot in the corridor, on a scale of $1 - 10$.

In the tests run in the corridor domain, both splitting criteria behaved similarly. The sum-squared error testing is significantly faster however.

In the Hexagonal Soccer (Hexcer) domain we performed empirical comparisons along two different dimensions – how fast does each algorithm learn to play, and to what level of expertise does the algorithm learn to play, discounting a “reasonable” initial learning period? Essentially we have two points on a learning curve.

In this experiment the pair of algorithms played each other at Hexcer for 1000 games. Wins were recorded for the first 500 games and the second 500 games. The first 500 games allowed us to measure learning speed as the agents started out with no knowledge of the game. The second 500 games gave an indication of the level of ability of the algorithm after the initial learning period.

This 1000 game test was then repeated 20 times. The results shown in table 3.2 are the number of games (mean $\pm$ standard deviation) won by each algorithm. The percentage at the end of each row is the level of statistical significance of the difference.

Table 3.2 shows that Continuous U Tree performs better than Prioritized Sweeping. Interestingly, looking at the (often quite large) trees generated by Continuous U Tree we can see that it has managed to learn the concept of "opponent behind."

	Prioritized Sweeping		Continuous U Tree		Significance
First 500 games	175 ± 93	$35\% \pm 19\%$	325 ± 93	$65\% \pm 19\%$	1%
Second 500 games	197 ± 99	$39\% \pm 20\%$	303 ± 99	$61\% \pm 20\%$	5%
Total	372 ± 183	$37\% \pm 18\%$	628 ± 183	$63\% \pm 18\%$	1%

Table 3.2: Hexcer results: Prioritized Sweeping vs. Continuous U Tree

3.4 Discussion

Continuous U Tree learns to play Hexcer with much less world experience than Prioritized Sweeping. Despite requiring less data, the Continuous U Tree algorithm is slower in real-time than the prioritized sweeping algorithm for the domains tested. For domains where data can be expensive or time-consuming to gather, trading algorithm speed for efficient use of data is important. In this area Continuous U Tree has an advantage over traditional reinforcement learning algorithms.

The corridor domain makes clear a number of details about the algorithm. Firstly, our algorithm successfully reduced a continuous space down to a discrete space with only 32 states (see figure 3.2). The algorithm has managed to find the 30° and 70° cutoff points with reasonable accuracy - it does not try to represent the entire temperature range. Continuous U Tree does not make perfect splits initially, but introduces further splits to refine its discretization. A possible enhancement would be to have old splits reconsidered in the light of new data and optimize them directly.

Secondly, the algorithm has divided the length of the corridor up as much as possible. Unlike the temperature axis, where large numbers of different datapoints values are grouped in one state, every different corridor location is in a different state.

The location of the agent in the corridor is directly relevant to achieving the goal. The state values in a fully discretized world change along this axis. By comparison, the values of different points below 30° in the temperature range, but with the same corridor location, are equal. Once the agent is below 30° it only moves 10% of the time – no further information is needed.

Like U Tree, Continuous U Tree inherits from its decision tree background the ability to handle moderately high dimensional state spaces. The original U Tree work used this capability to partially remove the Markov assumption. As we were playing a Markov game we did not implement this part of the U Tree algorithm in Continuous U Tree although there is no reason why this could not be done.

3.5 Limitations

While Continuous U Tree is effective in the domains shown, it has a number of limitations. Many of these limitations fall into two categories: limitations caused by the fact that the value function is stored as a single value in a leaf and hence is flat across each abstract state, and limitations caused by the semantics of the abstract MDP.

The fact that leaves store only a single value is a limitation. If there is an underlying discrete MDP embedded in a continuous state space, then Continuous U Tree can represent it exactly. If the underlying dynamics are themselves continuous, for example a wheeled robot driving towards a goal at the end of a 1 dimensional corridor, then Continuous U Tree may not be able to represent the value function exactly. In this corridor example, the true value function is an exponential function decaying from the reward at the end of the corridor. That cannot be modelled exactly by a tree with flat leaves, although as the algorithm gathers more data it continues splitting and hence approximates the true value function with increasing accuracy. We do not attempt to remedy this problem in this thesis, but refer the reader to the related work in Chapter 6.

The second major class of limitations is caused by the semantics of the abstract MDP. In particular, the abstract states are assumed to have the Markov property, even though at the early stages of tree formation this assumption is grossly false. This has a number of effects which we demonstrate in the simple grid-world shown in Figure 3.3. There is a single reward - the grey square in the top right. The agent can move in the four cardinal directions. There is no penalty for running into walls, but the agent does not move in that case.

Let us first consider the initial split. Like all reinforcement learners, Continuous U Tree is going to have to explore to find the reward. All transitions into the reward are going to have high q values. All other transitions are going to have equal q values, regardless of whether they are blocked by a wall or not. The initial split divides either the top row or the right column from the grid-world, as these are the only places with different q values. It takes a large number of sampled transitions for these splits to become statistically significant as the splits are axis parallel. The different q values are only at one end of the row that will be split off – their difference is diluted by the q values in the rest of the row. This effect becomes worse in higher dimensions; it becomes harder and harder to find the initial split.

Once this first split has been introduced, the new abstract MDP is solved and all transitions that cross into another leaf have q values different from the abstract self-transitions. This means that another split can be introduced – parallel to the first split, one step over. Note that the $q(I, a)$ values in a leaf only really contain information near the edges of that leaf. Splitting continues in this way, chopping layers from the edges, until the entire state is discretized.

Slicing from the edges of leaves in this way is bad. We noted above that we would end up having to discretize the entire space. We now can see that we are not even discretizing the space so that the intermediate results are as useful as possible.

Finally, we note that because much of the information in the q values is only visible one step away from another leaf, there is no information available for temporal abstraction which, by definition, requires multiple steps.

We note two possible directions to move to solve these problems. Firstly, introducing non-flat, usually linear, value functions into the leaves attacks many of these issues. It causes q values on abstract self-transitions to have values that depend upon the dynamics of the state space as well as the reward, making it easier to find splits. While we investigated this path, we found no techniques more effective than those discussed in the related work in Chapter 6, (*e.g.* Munos and Moore, 1999). Moving Continuous U Tree to have linear leaves does not allow the use of the types of temporal abstraction we propose in this thesis.

The approach we used was to try and fix the semantics of the abstract MDP. We discuss this work in Chapter 5.

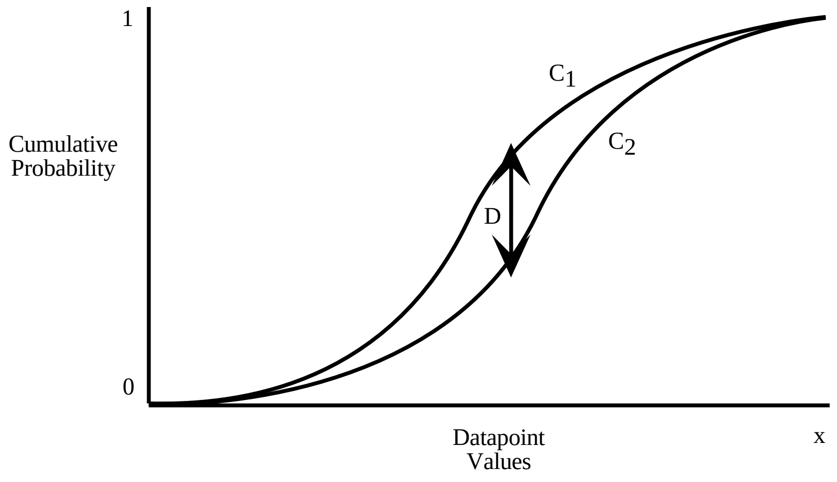


Figure 3.1: Two cumulative probability distributions

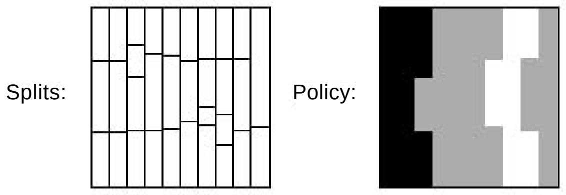


Figure 3.2: The learnt discretization and policy for the corridor task after 3000 steps. The y axis is the robot's temperature. The x axis is the location along the corridor. The goal is on the right hand edge. The policy was to move right at every point. Black areas indicate the heater was to be active. White areas indicate the cooler was to be active.

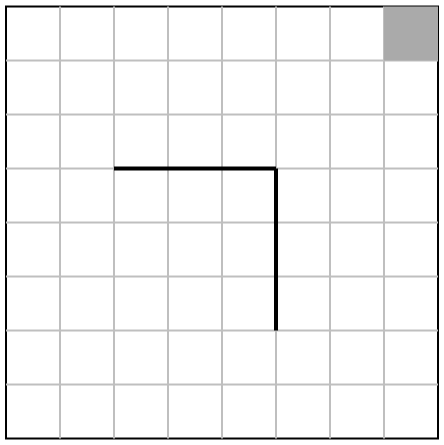


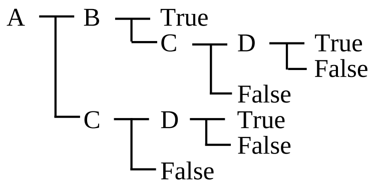
Figure 3.3: A simple gridworld to demonstrate some limitations of Continuous U Tree.

Chapter 4

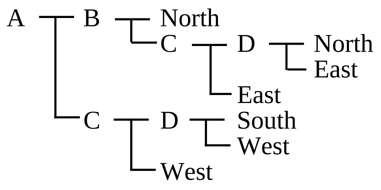
Lumberjack

Trees have been used for the representation of induced concepts in numerous areas of AI. In this thesis we are using trees to represent a discretization of a state space when solving Semi-Markov Decision Problems. Supervised learning is another area with large application of trees (Breiman et al., 1984; Quinlan, 1992). Despite their frequent use, trees are not perfect; to represent some concepts trees may need to represent some subconcepts multiple times. For example, to represent the boolean concept $AB \vee CD$ a decision tree has to repeat the representation of either AB or CD (see Figure 4.1a where CD is repeated).

This repetition of entire subtrees is well known in the supervised learning community and has been studied by several researchers (see Section 4.1). In addition, we have found that in most reinforcement learning domains in which temporal abstraction is useful, representing the policy as a tree repeats internal structure. Reversing our viewpoint, we can look at repeated structures in the tree representation of a policy as subtasks in that domain. For example, consider a concept mapping a tuple of boolean inputs, $\langle A, B, C, D \rangle$, to action



(a) A tree representing the boolean concept $AB \vee CD$



(b) A concept with repeated internal structure

Figure 4.1: Two trees with repeated structure

outputs, $\{\text{North}, \text{South}, \text{East}, \text{West}\}$ as shown in Figure 4.1b. The internal structure of the CD subtree is repeated even though the leaves are not. It chooses between either North and East, or South and West depending upon the value of A.

In most inductive systems work must be performed to learn each part of the tree. If a subconcept is represented twice then it must be learnt twice. Moreover, each individual representation of a subconcept is learnt using only part of the available data. For example, in Figure 4.1b the representation of CD when A is true is learned separately from the representation of CD when A is false.

One way to avoid this re-learning is to use a different representation. The target concept can be described as a hierarchy of concepts, each of which can use concepts below it in the hierarchy as building blocks. In the example above, the concept CD could be learned once and then simply referenced in multiple places while learning the full concept $AB \vee CD$.

In this chapter we present a new representation, the *linked decision forest*[1]. This representation allows trees in the forest to reference other trees in the forest as subconcepts. The linked decision forest does not have to repeatedly represent, and so repeatedly relearn, subconcepts.

We also introduce an algorithm, Lumberjack, for growing these linked forests. In this algorithm new trees are introduced and old trees removed as the algorithm progresses. Additionally, all trees in the forest are grown in parallel. This allows the representation used by the trees, which includes the other trees, to change dynamically. We show in the empirical results at the end of the chapter that Lumberjack generalizes more effectively than a simple decision tree on hierarchically decomposable concepts.

The hierarchical decomposition of the tree into a linked decision forest is presented in this chapter in terms of supervised learning. Given a series of examples from a function, we try to learn a representation that can predict the output for future inputs. It is important to note however, that the decomposition itself can be applied to trees in other domains.

This algorithm follows the *Minimum Description Length* (*MDL*) (Rissanen, 1983) or *Minimum Message Length* (*MML*) (Wallace and Boulton, 1968) paradigm. In this paradigm, a theory is encoded and then the data is encoded using the theory. The theory that gives the shortest combined code length is chosen. The theory is used to compress the data.

[1]The term ‘decision forest’ has been used previously in the machine learning literature to refer to a collection of different decision trees, each separately representing the same concept (Murphy and Pazzani, 1994). We introduce the term *linked decision forest* to refer to a collection of decision trees with references between the trees so the forest as a whole, not just the individual trees, represents a concept.

In Lumberjack the theory is itself compressed by extracting redundant subtrees and only representing them once. This extracted structure is itself represented as a tree and so the extraction algorithm can be run recursively. The result is a hierarchy of concepts that are used to represent the data.

4.1 Related Work in Substructure Detection

Duplicated subtrees, as in Figure 4.1a, are a well known problem. Two decision tree-like systems that attempt to factor out repeated substructure are the FRINGE system (Pagallo and Haussler, 1990) and the decision graph induction system of Oliver and Wallace (1992). Read once oblivious decision graphs (Kohavi, 1995) are also related, although they use a significantly different method to generate the graph.

The FRINGE system works by first growing a normal decision tree. Once this tree is fully grown, the last two decisions above each leaf in the tree (the fringe of the tree) are processed to form new attributes. The original tree is discarded and a new tree is grown using both the original attributes and the new attributes. The whole process is repeated, with the number of attributes constantly growing, until accuracy on a separate dataset starts dropping. The fact that attributes are not removed if they turn out not to be useful is an efficiency concern, as is the repeated re-growing of the tree.

Oliver and Wallace (1992) inferred decision graphs directly using the Minimum Message Length Principle (MML) (Wallace and Boulton, 1968; Quinlan and Rivest, 1989; Wallace and Patrick, 1993). The system proceeds much as would a decision tree learner, except for two changes. Instead of a depth first approach to recursively splitting the dataset, the splits are introduced in a best first manner; the location of the next decision node is chosen using MML. Also, instead of introducing a new decision node, the system can join two leaves together, forming a directed acyclic graph. This decision is also made using the MML criterion.

The HOODG (Kohavi, 1995) system is very closely related to the Ordered Binary Decision Diagrams of Bryant (1992). These have a number of differences from arbitrary decision graphs. Both HOODG and OBDDs require an ordering among the variables and only generate graphs that test the variables in that order. As discussed by Kohavi, this limits the representation so that it is less efficient than an arbitrary decision graph. Also, the algorithm is not incremental and so cannot be transferred to RL using the techniques described in Chapter 3. Hoey et al. (1999) have used Arithmetic Decision Diagrams in a reinforcement learning setting. In that work they are used in a non-approximate manner.

Both Oliver and Wallace (1992) and Kohavi (1995) use a decision graph representation. A decision graph is not capable of factoring out structure which is only repeated internally, like the CD subtree in Figure 4.1b. Additionally, the decision graph algorithm of Oliver and Wallace (1992) chooses when to factor out repeated structure (join two leaves) using MML. The algorithm is choosing subtrees to 'join' based on comparison of their outputs, without any comparison of the structures required to represent the correct subconcepts (which have not been grown at the time the decision to join is made). At the time the research in this chapter was performed we were intending to apply these algorithms to approximation of the value function of a reinforcement learning problem. When approximating the value function it is important to allow variation in the absolute value of a region of space while still being able to capture repeated structure. With our current approach, extracting structure from the *policy*, this need to extract internal structure is less urgent. It might be useful in future work to compare these algorithms for finding repeated structure.

Our work is based on the SEQUITUR algorithm (Nevill-Manning, 1996; Nevill-Manning and Witten, 1997) for the automatic decomposition of strings. Given a linear sequence of symbols with no prior structure, SEQUITUR forms a simple grammar where repeated substrings are factored out. For example, given the string $S \rightarrow abcdababcd$, SEQUITUR produces the grammar: $A \rightarrow ab, B \rightarrow Acd, S \rightarrow BAB$. This grammar re-represents the original string in a compact form.

It is important to note that finding the most compressive decomposition of this type for a linear string is an NP-hard problem (Storer, 1982). The problem in strings is reducible to the similar problem in trees[2], so decomposing trees for optimal compression is also NP-hard. SEQUITUR is a linear time heuristic algorithm for decomposing strings that has been shown to give good results.

4.2 The Linked Forest Representation

For linear strings the grammar is a well known representation for a hierarchical decomposition. We introduce the linked forest representation which allows hierarchical decomposition of trees. A linked forest is composed of trees with references between them in the same way a grammar is composed of rewrite rules with references between them. One tree

[2]A string can be embedded in a degenerate binary tree that only has non-leaf children on one side.

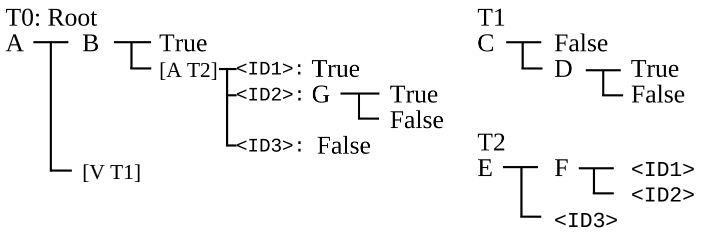


Figure 4.2: A linked decision forest showing the root tree T0, and the trees T1 and T2; T0 includes a value reference to T1, [V T1], and an attribute reference to T2, [A T2]

in the linked forest is marked as the root tree. The root node of this tree is the starting point for classification by the forest. Figure 4.2 shows an example of a boolean linked decision forest.

The inter-tree references take two forms. When a node makes a *value reference* to another tree the semantics are similar to a jump instruction; processing simply continues in the new tree. When a node makes an *attribute reference* to another tree the semantics are similar to a function call. The referencing node has children which are in one-to-one correspondence with the leaves of the referenced tree. Control is passed across to the referenced tree until a leaf is reached, then passed back to the corresponding child of the referencing node.

If a tree is only referenced by attribute references, an *attribute tree*, then it does not require class labels or other data in its leaves. It simply has ID values that allow the corresponding children to be found. Lumberjack does not yet form value trees. They are mentioned for comparison purposes. One can view FRINGE as forming attribute trees, like Lumberjack, and the Decision Graph induction algorithm as forming value trees.

4.3 The Lumberjack Algorithm

The SEQUITUR algorithm detects common subsequences in strings by tracking digrams. For our algorithm we define a structure similar to a digram for trees, a *di-node,* that can be hashed for fast duplicate detection.

A di-node is defined as a pair of internal nodes in the forest such that one node is a child of the other. Two di-nodes are defined to be equal if the parent nodes are equal, the child nodes are equal, and the child is in the same location in both di-nodes (*i.e.* child ordering is important). For example, in Figure 4.1a the two nodes labelled A and B form a di-node. The two nodes labelled A and B in Figure 4.2 also form a di-node. These di-nodes are equal; the difference in nearby nodes is irrelevant. There are also two di-nodes made up of nodes labelled C and D in Figure 4.1a. Those di-nodes are equal, but they are not equal to the di-node made up of nodes labelled C and D in Figure 4.2; the D node is not in the same location relative to its parent.

Note that either, or both, of the nodes in a di-node can be a reference to another tree, and so a di-node can represent an arbitrarily large set of nodes. Also note that two di-nodes are equal if and only if they represent equivalent sets of nodes that have been decomposed in the same way. In addition, note that matching di-nodes do not have to occur in the root tree, or even the same tree.

We are now in a position to give an overview of the Lumberjack algorithm. Table 5 shows the algorithm in detail. Initially the forest starts as a single tree with a single leaf node. Leaves are then split one at a time; they are replaced with new decision nodes. As the forest is updated a hash table records all di-nodes currently in the forest. We use the Minimum Description Length (MDL) principle to choose the next decision node and to decide when to stop growing the forest (the details of the MDL selection are discussed later). Once an internal node has been added the forest is checked for duplicate di-nodes using the di-node hash. Any non-overlapping duplicates are extracted to form a new attribute tree and the original di-nodes are replaced with references to the new tree. Any trivial attribute trees (trees referenced only once or having less than two internal nodes) are removed and their structure reinserted into the referencing tree(s).

This extraction and reinsertion of di-nodes removes all duplicated substructure from the forest. Because duplicate di-nodes are detected in all trees, it is common for the structure to be more than two levels deep.

Note that we only form attribute trees from internal nodes. When generating the grammar for a string it is possible to form a rewrite rule containing the last character of a string because the end of the string is unique. If we want to add any further characters we know where to add them. If we merge leaves in Lumberjack then we lose the ability to differentiate the positions where we might wish to add further nodes. While this might sometimes be useful for linked decision forests, as shown by Oliver and Wallace (1992), it is difficult to find a suitable criterion for doing this while retaining the ability to form attribute references.

Procedure 5: The main linked forest learning algorithm

```
1: $Forest \leftarrow$ new leaf, $Hash \leftarrow \emptyset$, $DAG \leftarrow \emptyset$
2: **repeat**
3:   $BestLength_{all} \leftarrow Length(Forest)$
4:   $BestLength_{new} \leftarrow \infty$
5:   **for all** leaves, $l$, in $BestForest$ **do**
6:     **if** splitting this leaf would mean splitting a non-leaf elsewhere **then**
7:       Continue with next leaf
8:     **end if**
9:     **for all** decision types, $d$ **do**
10:      **if** Introducing a decision of this type at this leaf would cause a cycle in $DAG$ **then**
11:        Continue with next decision type
12:      **end if**
13:      $Forest \leftarrow BestForest_{all}$
14:      Replace $l$ in $Forest$ with an internal node, $i$, of type $d$ and new leaves (clones of $l$)
15:      Update forest structure (see Procedure 6)
16:      $ThisLength \leftarrow Length(BestForest)$
17:      **if** $ThisLength < BestLength_{new}$ **then**
18:        $BestLeaf \leftarrow l$
19:        $BestDecision \leftarrow d$
20:        $BestLength_{new} \leftarrow ThisLength$
21:      **end if**
22:      Remove $i$ and replace it with $l$ {Undoing this change to the forest}
23:      Update forest structure (see Procedure 6)
24:    **end for**
25:  **end for**
26:  Update forest structure (see Procedure 6)
27:  **if** $BestLength_{new} < \infty$ **then**
28:    Replace $BestLeaf$ in $Forest$ with an internal node, $i$, of type $BestDecision$ and new leaves
       (clones of $BestLeaf$)
29:    **if** the parent of $i$ is itself a decision node **then**
30:      Add the di-node $\langle \text{parent}(i), i \rangle$ to $Hash$
31:      Update substructure (see Procedure 7)
32:    **end if**
33:  **end if**
34: **until** $BestLength_{all} < Length(Forest)$
35: Return $Forest$
```

Procedure 6: The subroutine to update forest structure

```
1: for all trees $t \in DAG$, in topological order do
2:   for all nodes $n \in t$, in post-order do
3:     if $n$ is a reference to another tree, $t_r$ then
4:       for all children $c$ of $n$ do
5:         if $c$ corresponds to a leaf in $t_r$ that no longer exists then
6:           Remove $c$ from the tree
7:         end if
8:       end for
9:       for all leaves $l \in t_r$ without children in $n$ do
10:        add a new leaf to $n$ as the corresponding child to $l$
11:      end for
12:    end if
13:  end for
14: end for
```

Procedure 7: The subroutine to find common substructure

```
1: while there are duplicate di-nodes, single use trees or degenerate trees do
2:   if there are duplicate di-nodes then
3:     Use the first pair of non-overlapping duplicate di-nodes to form a new attribute
       tree
4:     Replace the di-nodes with references to the new tree
5:   else if there are single use trees then
6:     Reinsert the tree
7:   else if there are degenerate trees then
8:     Reinsert the tree
9:   end if
10:  Update $Hash$ and $DAG$
11: end while
```

4.3.1 Altering the inductive bias

In the previous text we did not supply all the details of the algorithm. If the decision criteria for new nodes are chosen from only the original attributes, and only leaves of the root tree are extended, then the concept learned is the same as that learned by a normal tree induction algorithm; there is no change in inductive bias. The new representation has all repeated structure separated into other trees, but this is only a change in representation, not concept. We can change the inductive bias of the algorithm, and hence the concept learned, by extending the ways the forest is grown.

The first change is to allow the induction algorithm to split not only on the original attributes, but also to introduce an attribute reference to any tree in the forest. This can be viewed as a form of macro replay. The one restriction is that the use of this tree not introduce a cycle in the forest. Lumberjack records which trees reference which other trees in a directed acyclic graph (DAG). No split that would introduce a cycle in this graph is allowed.

The second change is to allow the algorithm to refine the attribute trees: we allow the induction algorithm to grow the forest not only at leaves of the root tree, but at the leaves of any tree in the forest. This can be viewed as a form of macro refinement. Again there is a restriction. Recall that leaves of attribute trees correspond with the children of the nodes that reference them. If you split a leaf of an attribute tree, then you must split the corresponding children of the referencing node(s). If any of the corresponding children is not a leaf then we do not allow the split.

Growing attribute trees changes the number of outcomes of decision nodes elsewhere in the forest. That in turn changes the number of outcomes of other decision nodes, *etc.* Because the trees form a DAG, it is possible to update the trees in reverse topological order and know that all trees being referenced by the tree currently being updated are themselves up to date.

Finally, the correspondence between the leaves of an attribute tree and the children of a node that references that tree is important for the hashing of di-nodes. The hash table should use that correspondence rather than child numbering for generating hash codes and testing equality. By avoiding the use of child numbering the algorithm does not have to re-hash di-nodes when an attribute tree grows or shrinks.

4.3.2 Example: Growing a CNF function

Figures 4.3 and 4.4 show two series of decision forests that might be generated while growing the boolean function $ABC \vee DEF$. The algorithm is deterministic and so to generate two different forests would require different sets of training data, even if sampled from the same original concept. Rather than use real data and MDL we have chosen the decision nodes added at each stage ourselves to demonstrate aspects of the algorithm. To save space a number of steps are omitted.

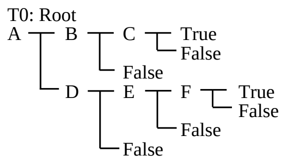
Figure 4.3 demonstrates the common substructure detection of Lumberjack. In part (a) the forest is a single tree with no repeated structure. In part (b) we show an intermediate stage with a repeated di-node, DE. This is then extracted in part (c) to form a new tree, T1, with attribute references in the root tree. Note that internal structure has been extracted; the second reference to T1 has a non-leaf child, F, through outcome `<ID1>`.

Parts (d) and (e) show another aspect of the decomposition. Part (d) shows the root tree with a repeated di-node, two copies of the [A T1] F combination mentioned above. In part (e) these di-nodes have been extracted to form a new tree, T2. Syntactically, the repeated structure was constant size and so could be detected quickly. However, because repeated references match, semantically the repeated structure was a larger subtree.

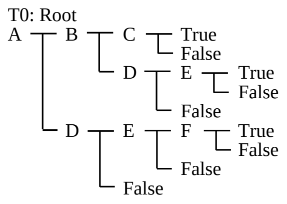
In part (e), T1 is only referenced once. In this case Lumberjack reinserts the tree. This is shown in part (f). This reinsertion is important for the algorithm. As noted above, two di-nodes match only if they represent the same structure decomposed in the same way. Reinsertion reduces the number of ways a concept can be decomposed and so removes a barrier to matching equivalent subtrees. Reducing the number of trees in the forest also reduces the number of possible decision nodes that could be introduced and so increases the speed of the algorithm.

Figure 4.4 again shows a set of forests that could be generated while learning the function $ABC \vee DEF$. One can assume this was learnt from a different dataset to the example in Figure 4.3. Again, rather than use real data and MDL we have chosen the decision nodes added at each stage ourselves to demonstrate aspects of the algorithm. Here we show Lumberjack reusing and refining previously learnt subconcepts.

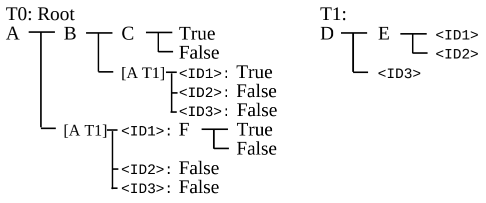
Again, we skip some normal growth steps and start following in detail when the tree in (a) has been learnt. We will then assume our data causes DE to be grown in another subtree leading to the forest in (b). It is at this point that the first repeated di-node, DE, is detected. The di-node is extracted to form a new tree in part (c).



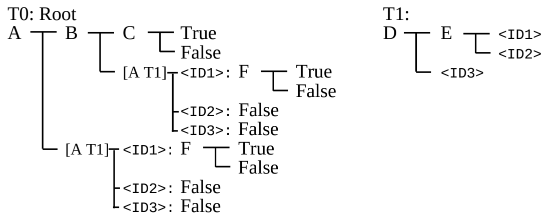
(a) A tree representing part of the concept



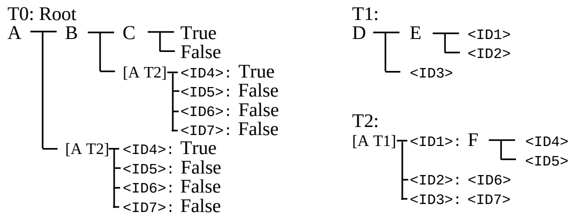
(b) The tree with a duplicate di-node, DE



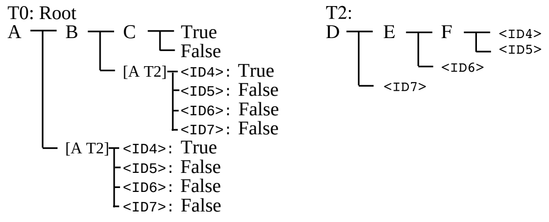
(c) A forest where the duplicate di-node from (b) has been separated into T1



(d) A forest with a duplicate di-node, [A T1] F.



(e) The duplicate from (d) has been removed to form T2



(f) Tree T1 from (e) has been re-inserted into T2

Figure 4.3: A series of forests while learning the boolean function $ABC \vee DEF$. This series demonstrates the common substructure detection aspects of Lumberjack.

Node Type	Bits
Leaf	$\log_2(\frac{b}{b-1})$
decision node	$\log_2(b) + \log_2(N_A + N_{PT})$

Table 4.1: Costs to encode a non-root node

Having found the substructure we can then immediately use it by splitting on the new tree. The resulting forest is shown in part (d).

Finally we can grow the forest at the leaves of any tree. In this case we grow tree T1 so that it represents the concept DEF (see part (e)). This involves removing leaf `<ID1>` and replacing it with a decision and two new leaves. Note that in each of the references to T1 all the children through `<ID1>` were leaves. These are removed and new children added for the new IDs. Children through other IDs are unchanged.

4.3.3 MDL coding of linked decision forests

The Minimum Description Length (Rissanen, 1983) or Minimum Message Length (Wallace and Boulton, 1968) Principle is a way of finding an inductive bias. It uses Bayes' Rule[3] and Shannon's information theory[4] to choose between competing models for data. The model and data are both encoded according to a coding scheme. The model which gives the shortest total code length is chosen.

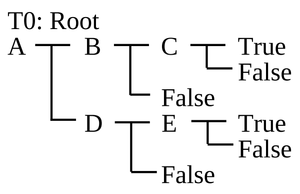
The Lumberjack algorithm could also be used with other decision node selection criteria. MDL was chosen for ease of implementation and because it supplies a stopping criterion.

Our coding scheme for MDL comparisons is a minor change from the Wallace and Patrick (1993) scheme for decision trees. Let N_T be the number of trees, N_A the number of attributes, N_{PT} the number of trees after the current tree in the topological ordering and b be the branching factor of our parent node. First, the number of trees in the forest is encoded using $L^*(N_T)$ bits.[5] Then the trees are encoded in reverse topological order. Each tree is encoded by performing a pre-order traversal of the tree and encoding each node using the number of bits shown in Table 4.1.

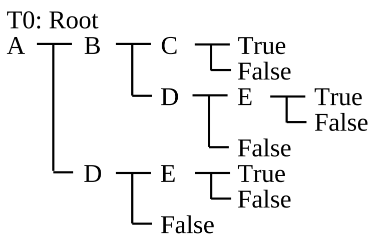
[3]We use a weak form of Bayes' Rule: $P(T|D) \propto P(T)P(D|T)$.

[4]The optimal code length in bits of a symbol that has probability p is $-\log_2(p)$.

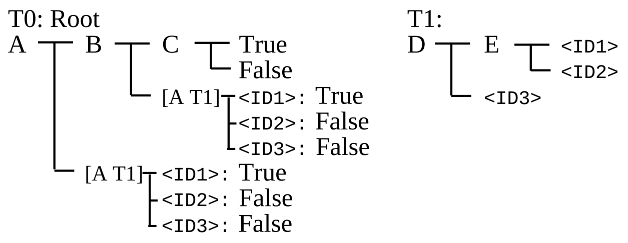
[5]$L^*(X) = \log_2^*(X) + \log_2(c)$, where $c \simeq 2.865064$, is a code length for an arbitrary integer. $\log_2^*(X) = \log_2(X) + \log_2(\log_2(X)) + \ldots$ summing only positive terms (Rissanen, 1983).



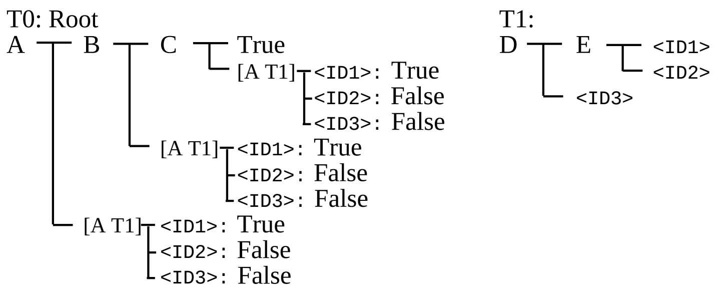
(a) A tree representing part of the concept



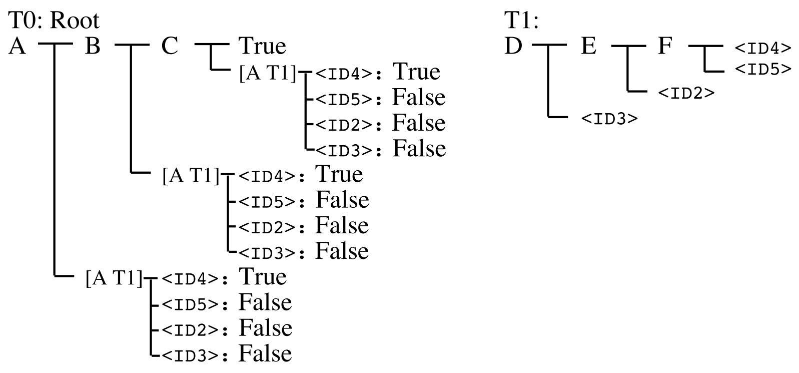
(b) The tree with a duplicate di-node, DE



(c) A forest where the duplicate di-node from (b) has been separated into T1



(d) Tree T1 from (c) has been reused



(e) Tree T1 from (d) has been extended

Figure 4.4: A series of forests while learning the boolean function $ABC \vee DEF$. This series demonstrates the aspects of Lumberjack that result in a change of bias.

- For each value leaf in the forest
  - Encode each example using $-\log_2(p_{i,j})$ bits

where,

- i is the number of examples of this class seen so far in this leaf
- j is the total number of examples seen so far in this leaf
- M is the number of classes
- $p_{i,j} = \frac{i+1}{j+M}$

Table 4.2: Costs of MDL example coding

The one cost not yet specified is the cost to encode the root nodes of the trees. These have no parent node and hence b is undefined. When there is only one tree, a leaf at the root is encoded using $\log_2(N_A)$ bits and a decision node is encoded using $\log_2(\frac{N_A}{N_A-1})+\log_2(N_A)$ bits, as in Wallace and Patrick (1993). When there is more than one tree, we know that none of the root nodes are leaves. The root decision nodes can be encoded using only $\log_2(N_A + N_{PT})$ bits. Finally, the examples are encoded using the costs in Table 4.2.

4.4 Experiments

We tested Lumberjack using standard supervised learning experiments. We compared the generalization accuracy of a decision tree learner and Lumberjack, each using the same MDL coding. The results are shown in Figures 4.5 and 4.6. The graphs show the averages over 10 trials. We tested for significance using a paired Wilcoxon rank-sum test ($p = 0.05$).

The first set of results are for the boolean function $ABC \vee DEF \vee GHI$. Training samples were sampled with replacement from the concept, then the output was flipped in 10% of the samples. The testing dataset was a complete dataset without noise. Results are shown in Figure 4.5. The difference in error rate between the tree and forest algorithms is significant for sample sizes 1000 through 2500 inclusive, and also for the 3000 sample

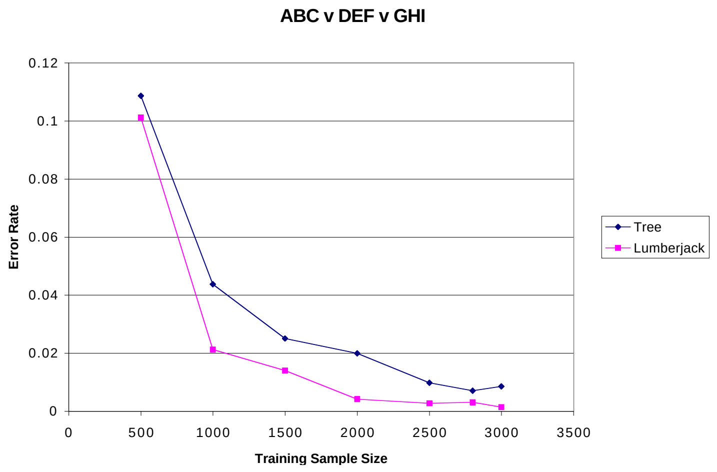


Figure 4.5: Experimental results learning the concept $ABC \vee DEF \vee GHI$

dataset.[6]

The second set of results uses a dataset generated by mapping a reinforcement learning problem back into a supervised learning problem. The domain used was the walking robot domain described in Section 2.6.4. This problem was fed into a traditional Markov Decision Problem algorithm, and the resulting policy was used as a dataset for our supervised learning experiment. The domain is discrete. The Δs each have only 3 possible values, and the X, Y location was encoded using a series of variables $\{X < 1, X < 2, \ldots, X < 9, Y < 1, Y < 2, \ldots, Y < 9\}$. This coding is similar to the one implicitly used by C4.5 for continuous variables. Training datasets were generated by randomly sampling, with replacement, from this true dataset. The testing dataset was the full true dataset.

Again the results, in Figure 4.6, are the averages over 10 trials. There is a significant difference between the two algorithms for sample sizes of 2000 or more. While the results at sample size 4000 are still significant, it is clear that both algorithms are converging again as sample size increases; the tree has enough data to grow the repeated structure. Looking at the forest for large sample sizes it is possible to see the separation of structure representing the maze from structure representing the ability to walk.

[6]We also compared with C4.5. C4.5 is always significantly better than the MDL tree learning system. This is a well known deficiency of MDL vs. C4.5 and is orthogonal to the use of Lumberjack style decomposition. With 1000 or more datapoints, C4.5 and Lumberjack perform similarly.

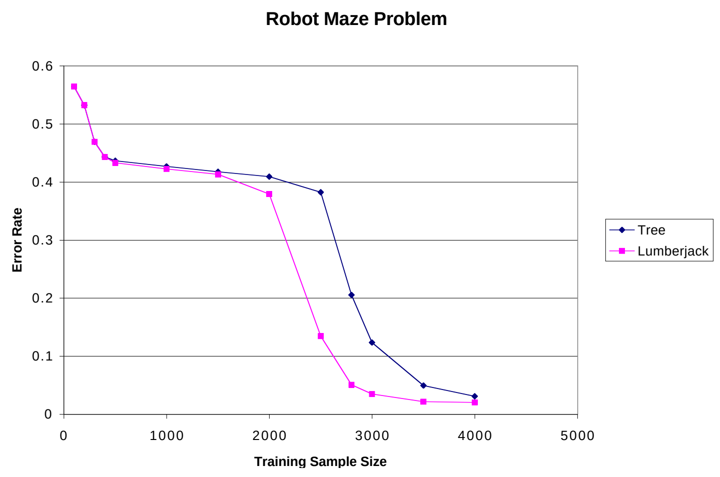


Figure 4.6: Experimental results learning a policy to walk through a maze

4.5 Lumberjack and Solving SMDPs

In the introduction to this chapter we noted that although this chapter was phrased in terms of supervised learning, the decomposition could be applied to other domains. In particular we have applied lumberjack style decomposition directly to the state tree in the Continuous U Tree algorithm as that tree is being grown. This does not work particularly well. The reason appears to be the nonstationary nature of the learning that occurs in Continuous U Tree – the dataset being used to divide the state space changes each time the MDP is re-solved. However, as shown above, Lumberjack is effective in decomposing a reinforcement learning policy once the problem has been solved and the policy has become stationary.

Chapter 5

TTree

In this chapter we introduce the Trajectory Tree, or TTree, algorithm. The goal of this algorithm is to take an SMDP and a small collection of supplied policies, and discover which supplied policy should be used in each state to solve the SMDP. We wish to do this in a way that is more efficient than finding the optimal policy directly.

This is part of our approach to solving a series of SMDPs that was introduced in Chapter 1. To recap, our approach consists of two stages: firstly, the solutions to early problems are processed. In the second stage the results of that processing are used to solve later problems. In this chapter we describe the second stage of that process in more detail. The input to this algorithm is a set of policies that are likely to be useful for solving sub-problems in the current problem.

The TTree algorithm is an extension of the Continuous U Tree algorithm described in Chapter 3. In addition to adding the ability to use temporal abstraction, we also fix some of the semantic issues discussed in Section 3.5.

TTree uses policies as temporally abstract actions. They are solutions to subtasks that we expect the agent to encounter. We refer to these supplied policies as abstract actions to distinguish them from the solution – the policy we are trying to find. This definition of “abstract actions” is different from previous definitions. Other definitions of abstract actions in reinforcement learning have termination criteria that our definition does not. Definitions of abstract actions in planning (*e.g.* Knoblock, 1991), where an abstract action is a normal action with some pre-conditions removed, are even further removed from our definition. This ‘planning’ definition of abstract actions is closer to the concept of state abstraction than temporal abstraction.

Each of the supplied abstract actions is defined over the same set of base-level actions as the SMDP being solved. As a result, using the abstract actions gives us no more representational power than representing the policy through some other means, *e.g.* a table. Additionally, we ensure that there is at least one abstract action that uses each base-level action in each state, so that any policy for the SMDP can be represented as as a policy over the abstract actions.

The previous paragraph shows that a policy over the abstract actions has identical representational power to a normal policy over the states of an SMDP. However, if we have a policy mapping abstract states to abstract actions, then we have increased the representation power over a policy mapping abstract states to normal actions. This increase in power allows our abstract states to be larger while still representing the same policy.

5.1 Definitions

In Chapter 2, equations 2.12 and 2.13, we gave the definition of the Q and value functions of an SMDP. The $Q^\pi(s,a)$ function is the expected sum of discounted reward for taking action a in state s and following the policy π from then onwards. We reprint the equations here:

$$Q^\pi(s,a) = R(s,a) + \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t V^\pi(s')\,\mathrm{d}t \tag{5.1}$$

$$V^\pi(s) = Q^\pi\left(s, \pi(s)\right) \tag{5.2}$$

We now introduce a related function, the T^π function. This function is defined over a set of states $\mathcal{S}' \subset \mathcal{S}$. It measures the discounted sum of reward for following the given action until the agent leaves $\mathcal{S}'$, then following the policy π.

$$\begin{aligned} T^\pi_{\mathcal{S}'}(s,a) = R(s,a) & \\ &+ \sum_{s' \in \mathcal{S}'} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t T^\pi_{\mathcal{S}'}(s',a)\,\mathrm{d}t \\ &+ \sum_{s' \in (\mathcal{S} - \mathcal{S}')} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t V^\pi(s')\,\mathrm{d}t \end{aligned} \tag{5.3}$$

We assume that instead of sampling P and R directly from the world, our agent instead samples from a *generative model* of the world, *e.g.* Ng and Jordan (2000). This is a function, $G : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \Re \times \Re$, that takes a state and an action and returns a next state, a time and a reward for the transition. Our algorithm uses G to sample trajectories through the state space starting from randomly selected states.

5.2 The TTree Algorithm

TTree works by building an abstract SMDP that is smaller than the original, or base, SMDP. The solution to this abstract SMDP is an approximation to the solution to the base SMDP. The abstract SMDP is formed as follows: The states in the abstract SMDP, the *abstract states*, are formed by partitioning the states in the base SMDP; each abstract state corresponds to the set of base level states in one element of the partition. Each base level state falls into exactly one abstract state. Each action in the abstract SMDP, an *abstract action*, corresponds to a policy, or stochastic policy, in the base SMDP. We allow these abstract actions to have stochastic outcomes. The abstract transition and reward functions are found by sampling trajectories from the base SMDP.

We introduce some notation to help explain the algorithm. We use a bar over a symbol to distinguish the abstract SMDP from the base SMDP, *e.g.* $\bar{s}$ vs. s, or $\bar{\mathcal{A}}$ vs. $\mathcal{A}$. This allows us a shorthand notation: when we have a base state, s, we use $\bar{s}$ to refer specifically to the abstract state containing s. Also, when we have an abstract action $\bar{a}$ we use $\pi_{\bar{a}}$ to refer to the base policy corresponding to $\bar{a}$ and hence $\pi_{\bar{a}}(s)$ is the corresponding base action at state s. Additionally, we sometimes overload $\bar{s}$ to refer to the set of base states that it corresponds to, *e.g.* $s \in \bar{s}$. Finally, it is useful, particularly in the proofs, to define functions that describe the base states within an abstract state, $\bar{s}$, but only refer to abstract states outside of $\bar{s}$. We mark these functions with a tilde. For example, we can define a function related to $T_{\mathcal{S}'}(s, a)$ above, $\tilde{T}_{\bar{s}}(s, a)$.

$$
\begin{aligned}
\tilde{T}_{\bar{s}}(s, a) = R(s, a) \\
&+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{T}_{\bar{s}}(s', a)\, \mathrm{d}t \\
&+ \sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s'})\, \mathrm{d}t
\end{aligned}
\tag{5.4}
$$

Note that the $\tilde{T}_{\bar{s}}$ function is labelled with a tilde, and hence within the abstract state $\bar{s}$ we refer to base level states, outside of $\bar{s}$ we refer to the abstract value function over abstract states.

We describe the TTree algorithm from a number of different viewpoints. First we describe how TTree builds up the abstract SMDP, $\langle \bar{\mathcal{S}}, \bar{\mathcal{A}}, \bar{P}, \bar{R} \rangle$. Then we follow through the algorithm in detail, and finally we give a high level overview of the algorithm comparing it with previous algorithms.

5.2.1 Defining the abstract SMDP

TTree uses a tree to partition the base level state space into abstract states. Each node in the tree corresponds to a region of the state space with the root node corresponding to the entire space. As our current implementation assumes state dimensions are discrete, internal nodes divide their region of state space along one dimension with one child for each discrete value along that dimension. It is a small, but currently unimplemented, extension to handle continuous and ordered discrete attributes in the same way Continuous U Tree does. Leaf nodes correspond to abstract states; all the base level states that fall into that region of space are part of the abstract state.

TTree uses a set of abstract actions for the abstract SMDP. Each abstract action corresponds to a base level policy. There are two ways in which these abstract actions can be obtained; they can be supplied by the user, or they can be generated by TTree. In particular, TTree generates one abstract action for each base level action, and one additional 'random' abstract action. The 'random' abstract action is a base level stochastic policy that performs a random base level action in each base level state. The other generated abstract actions are degenerate base level policies: they perform the same base level action in every base level state: $\forall s; \pi_{\bar{a_1}}(s) = a_1, \pi_{\bar{a_2}}(s) = a_2, \ldots, \pi_{\bar{a_k}}(s) = a_k$. These generated abstract actions are all that is required by the proof of correctness. Any abstract actions supplied by the user are hints to speed things up and are not required for correctness.

Informally, the abstract transition and reward functions are the expected result of starting in a random base state in the current abstract state and following a trajectory through the base states until we reach a new abstract state. To formalize this we define two functions. $\tilde{R}_{\bar{s}}(s, \bar{a})$ is the expected discounted reward of starting in state s and following a trajectory through the base states using $\pi_{\bar{a}}$ until a new abstract state is reached. If no new abstract state is ever reached, then $\tilde{R}$ is the expected discounted reward of the infinite trajectory.

$\tilde{P}_{s,\bar{a}}(\bar{s'}, t)$ is the expected probability, over the same set of trajectories as $\tilde{R}_{\bar{s}}(s, \bar{a})$, of reaching the abstract state $\bar{s'}$ in time t. If $\bar{s'}$ is $\bar{s}$ then we change the definition; $\tilde{P}_{s,\bar{a}}(\bar{s'}, t)$ is the probability that the trajectory never leaves state $\bar{s}$ when $t = \infty$ and 0 otherwise.

We note that assigning a probability mass to $t = \infty$ is a mathematically suspect thing to do as it assigns a probability mass, rather than a density, to a single 'point' and, furthermore, that 'point' is ∞. We justify the use of $\tilde{P}_{s,\bar{a}}(\bar{s}, \infty)$ as a notational convenience for "the probability we never leave the current state" as follows. We note that each time P is referenced with $\bar{s'} = \bar{s}$, it is then multiplied by γ^t, and hence for $t = \infty$ the product is zero. This is the correct value for an infinitely discounted reward. In the algorithm, as opposed to the proof, $t = \infty$ is approximated by $t \in (\text{MAXTIME}, \infty)$. MAXTIME is a constant in the algorithm, chosen so that γ^{MAXTIME} multiplied by the largest reward in the SMDP is approximately zero. The exponential discounting involved means that MAXTIME is usually not very large.

The definitions of $\tilde{P}$ and $\tilde{R}$ are expressed in the following equations:

$$\tilde{R}_{\bar{s}}(s, \bar{a}) = R(s, \pi_{\bar{a}}(s)) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,\pi_{\bar{a}}(s)}(s', t) \gamma^t \tilde{R}_{\bar{s}}(s', \bar{a}) \, \mathrm{d}t \tag{5.5}$$

$$\tilde{P}_{s,\bar{a}}(\bar{s'}, t) = \begin{cases} \displaystyle\sum_{s'' \in \bar{s'}} P_{s,\pi_{\bar{a}}(s)}(s'', t) \\ \displaystyle\quad + \sum_{s'' \in \bar{s}} \int_{t'=0}^{t} P_{s,\pi_{\bar{a}}(s)}(s'', t') \tilde{P}_{s'',\bar{a}}(\bar{s'}, t - t') \, \mathrm{d}t' & : \bar{s'} \neq \bar{s} \\ 0 & : \bar{s'} = \bar{s}, t \neq \infty \\ \displaystyle 1 - \sum_{\bar{s''} \neq \bar{s}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s''}, t) \, \mathrm{d}t & : \bar{s'} = \bar{s}, t = \infty \end{cases} \tag{5.6}$$

Here $\tilde{R}$ is recursively defined as the expected reward of the first step plus the expected reward of the rest of the trajectory. $\tilde{P}$ also has a recursive formula. The first summation is the probability of moving from s to $\bar{s'}$ in one transition. The second summation is the probability of transitioning from s to another state $s'' \in \bar{s}$ in one transition, and then continuing from s'' on to $\bar{s'}$ in a trajectory using the remaining time. Note, the recursion in the definition of $\tilde{P}$ is going to be bounded as there are no zero time cycles in the SMDP (see Section 2.1).

We can now define the abstract transition and reward functions, $\bar{P}$ and $\bar{R}$, as the expected values over all base states in the current abstract state of $\tilde{P}$ and $\tilde{R}$:

$$\bar{P}_{\bar{s},\bar{a}}(\bar{s'},t) = \mathop{\mathrm{E}}_{s\in\bar{s}} \tilde{P}_{s,\bar{a}}(\bar{s'},t) \tag{5.7}$$

$$\bar{R}(\bar{s},\bar{a}) = \mathop{\mathrm{E}}_{s\in\bar{s}} \tilde{R}_{\bar{s}}(s,\bar{a}) \tag{5.8}$$

In english, $\bar{P}$ and $\bar{R}$ are the expected transition and reward functions if we start in a random base level state within the current abstract state and follow the supplied abstract action until we reach a new abstract state.

5.2.2 An overview of the TTree algorithm

In the algorithm $\bar{P}$ and $\bar{R}$ are not calculated directly from the above formulae. Rather, they are sampled by following trajectories through the base level state space as follows. A set of base level states is sampled from each abstract state. From each of these start states, for each abstract action, the algorithm uses the generative model to sample a series of trajectories through the base level states that make up the abstract state. In detail for one trajectory: Let the abstract state we are considering be the state $\bar{s}$. The algorithm first samples a set of base level start states, $\{s_0, s_1, \ldots, s_k\} \in \bar{s}$. It then gathers the set of base level policies for the abstract actions, $\{\pi_{\bar{a}_1}, \pi_{\bar{a}_2}, \ldots, \pi_{\bar{a}_l}\}$. For each start state, s_i, and policy, $\pi_{\bar{a}_j}$, in turn, the agent samples a series of base level states from the generative model forming a trajectory through the low level state space. As the trajectory progresses, the algorithm tracks the sum of discounted reward for the trajectory, and the total time taken by the trajectory. The algorithm does not keep track of the intermediate base level states.

These trajectories have a number of termination criteria. The most important is that the trajectory stops if it reaches a new abstract state. The trajectory also stops if the system detects a deterministic self-transition in the base level state, if an absorbing state is reached, or if the trajectory exceeds a predefined length of time, MAXTIME. The result for each trajectory is a tuple, $\langle s_{start}, \bar{a}_j, s_{stop}, t, r\rangle$, of the start base level state, abstract action, end base level state, total time and total discounted reward.

We turn the trajectory into a sample transition in the abstract SMDP: $\langle \bar{s}_{start}, \bar{a}_j, \bar{s}_{stop}, t, r\rangle$. The sample transitions are combined to estimate the abstract transition and reward functions, $\bar{P}$ and $\bar{R}$.

The algorithm now has a complete abstract SMDP. It can solve it using traditional techniques, *e.g.* Moore and Atkeson (1993), to find a policy for the abstract SMDP: a function from abstract states to the abstract action that should be performed in that abstract state. However, the abstract actions are base level policies, and the abstract states are sets of base level states, so we also have a function from base level states to base level actions; we have a policy for the base SMDP.

Having found a policy, TTree then looks to improve the accuracy of its approximation by increasing the resolution of the state abstraction. It does this by dividing abstract states; growing the tree. In order to grow the tree, we need to know which leaves should be divided and where they should be divided. A leaf should be divided when the utility of performing an abstract action is not constant across the leaf, or if the best action changes across a leaf.

We can use the trajectories sampled earlier to get point estimates of the T function defined in equation 5.3. First, we assume that the abstract value function, $\bar{V}$, is an approximation of the base value function, V. Making this substitution gives us the $\tilde{T}$ function defined in equation 5.4. The sampled trajectories with the current abstract value function allow us to estimate $\tilde{T}$. We refer to these estimates as $\hat{T}$. For a single trajectory $\langle s_i, \bar{a_j}, s_{stop}, r, t\rangle$ we can find $\bar{s}_{stop}$ and then get the estimate[1]:

$$\hat{T}_{\bar{s}}(s_i, \bar{a_j}) = r + \gamma^t \bar{V}(\bar{s}_{stop}) \tag{5.9}$$

From these $\hat{T}(s, \bar{a})$ estimates we obtain three different values used to divide the abstract state. Firstly, we divide the abstract state if $\max_{\bar{a}} \hat{T}(s, \bar{a})$ varies across the abstract state. Secondly, we divide the abstract state if the best action, $\mathrm{argmax}_{\bar{a}} \hat{T}(s, \bar{a})$, varies across the abstract state. Finally, we divide the abstract state if $\hat{T}(s, \bar{a})$ varies across the state for any abstract action. It is interesting to note that while the last of these criteria contains a superset of the information in the first two, and leads to a higher resolution discretization of the state space once all splitting is done, it leads to the splits being introduced in a different order. If used as the sole splitting criterion, $\hat{T}(s, \bar{a})$ is not as effective as $\max_{\bar{a}} \hat{T}(s, \bar{a})$ for intermediate trees.

Once a division has been introduced, all trajectories sampled from the leaf that was divided are discarded, a new set of trajectories is sampled in each of the new leaves, and the algorithm iterates.

[1]It has been suggested that it might be possible to use a single trajectory to gain $\hat{T}$ estimates at many locations. We are wary of this suggestion as those estimates would be highly correlated; samples taken from the generative model near the end of a trajectory would affect the calculation of many point estimates.

Constant	Definition
N_a	The number of trajectory start points sampled from the entire space each iteration
N_l	The minimum number of trajectory start points sampled in each leaf
N_t	The number of trajectories sampled per start point, abstract action pair
MAXTIME	The number of time steps before a trajectory value is assumed to have converged. Usually chosen to keep $\gamma^{\text{MAXTIME}} r/(1-\gamma^t) < \epsilon$, where r and t are the largest reward and smallest time step, and ϵ is an acceptable error

Table 5.1: Constants in the TTree algorithm

5.2.3 The TTree algorithm in detail

The TTree algorithm is shown in Procedure 8. The various constants referred to are defined in Table 5.1.

The core of the TTree algorithm is the trajectory. As described above, these are paths through the base-level states within a single abstract state. They are used in two different ways in the algorithm; to discover the abstract transition function and to gather data about where to grow the tree and increase the resolution of the state abstraction. We first discuss how trajectories are sampled, then discuss how they are used.

Trajectories are sampled in sets, each set starting at a single base level state. The function to sample one of these sets of trajectories is shown in Procedure 9. The set of trajectories contains N_t trajectories for each abstract action. Once sampled, each trajectory is recorded as a tuple of start state, abstract action, resulting state, time taken and total discounted reward, $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total} \rangle$, with s_{start} being the same for each tuple in the set. The tuples in the trajectory set are stored along with s_{start} as a sample point, and added to the leaf containing s_{start}.

The individual trajectories are sampled with the randomness being controlled (Ng and Jordan, 2000; Strens and Moore, 2001). Initially the algorithm stores a set of N_t random numbers that are used as seeds to reset the random number generator. Before the j^{th} trajectory is sampled, the random number generator used in both the generative model and any stochastic abstract actions is reset to the j^{th} random seed. This removes some of the

Procedure 8: Procedure $\mathrm{TTree}(\mathcal{S}, \bar{\mathcal{A}}, G, \gamma)$

```
1: $tree \leftarrow$ a new tree with a single leaf corresponding to $\mathcal{S}$
2: **loop**
3:    $\mathcal{S}_a \leftarrow \{s_1, \ldots, s_{N_a}\}$ sampled from $\mathcal{S}$
4:    **for all** $s \in \mathcal{S}_a$ **do**
5:       $\mathrm{SampleTrajectories}(s, tree, \bar{\mathcal{A}}, G, \gamma)$ {see Procedure 9}
6:    **end for**
7:    $\mathrm{UpdateAbstractSMDP}(tree, \bar{\mathcal{A}}, G, \gamma)$ {see Procedure 10}
8:    $\mathrm{GrowTTree}(tree, \bar{\mathcal{A}}, \gamma)$ {see Procedure 11}
9: **end loop**
```

randomness in the comparison of the different abstract actions within this set of trajectories.

There are four stopping criteria for a sampled trajectory. Reaching another abstract state and reaching an absorbing state are stopping criteria that have already been discussed. Stopping when MAXTIME time steps have passed is an approximation. It allows us to get approximate values for trajectories that never leave the current state. Because future values decay exponentially, MAXTIME does not have to be very large to accurately approximate the trajectory value (*e.g.* Ng and Jordan, 2000). The final stopping criterion, stopping when a deterministic self-transition occurs, is an optimization, but it is not always possible to detect deterministic self-transitions. The algorithm works without this, but samples longer trajectories waiting for MAXTIME to expire, and hence is less efficient.

The TTree algorithm samples trajectory sets in two places. In the main procedure, TTree randomly samples start points from the entire base level state space and then samples trajectory sets from these start points. This serves to increase the number of trajectories sampled by the algorithm over time regardless of resolution. Procedure 10 also samples trajectories to ensure that there sampled trajectories in every abstract state to build the abstract transition function.

As well as using trajectories to find the abstract transition function, TTree also uses them to generate data to grow the tree. Here trajectories are used to generate three values. The first is an estimate of the T function, $\hat{T}$, the second is an estimate of the optimal abstract action, $\hat{\pi}(s) = \mathrm{argmax}_{\bar{a}} \hat{T}(s, \bar{a})$, and the third is the value of that action, $\max_{\bar{a}} \hat{T}(s, \bar{a})$. As noted above, trajectories are sampled in sets. The entire set is used by TTree to estimate the $\hat{T}$ values and hence reduce the variance of the estimates.

Procedure 9: Procedure $\mathrm{SampleTrajectories}(s_{start}, tree, \bar{\mathcal{A}}, G, \gamma)$

```
1: Initialize new trajectory sample point, $p$, at $s_{start}$ $\{p$ will store $N_t$ trajectories for each of the $|\bar{\mathcal{A}}|$ actions$\}$
2: Let $\{\text{seed}_1, \text{seed}_2, \ldots, \text{seed}_{N_t}\}$ be a collection of random seeds
3: $l \leftarrow \mathrm{LeafContaining}(tree, s_{start})$
4: for all abstract actions $\bar{a} \in \bar{\mathcal{A}}$ do
5:     let $\pi_{\bar{a}}$ be the base policy associated with $\bar{a}$
6:     for $j = 1$ to $N_t$ do
7:         Reset the random number generator to seed$_j$
8:         $s \leftarrow s_{start}$
9:         $t_{total} \leftarrow 0$, $r_{total} \leftarrow 0$
10:        repeat
11:            $\langle s, t, r \rangle \leftarrow G(s, \pi_{\bar{a}}(s))$
12:            $t_{total} \leftarrow t_{total} + t$
13:            $r_{total} \leftarrow r_{total} + \gamma^{t_{total}} r$
14:        until $s \notin l$, or $t_{total} >$ MAXTIME, or
           $\langle s', *, * \rangle = G(s, \pi_{\bar{a}}(s))$ is deterministic and $s = s'$, or $s$ is an absorbing state
15:        if the trajectory stopped because of a deterministic self transition then
16:            $r_{total} \leftarrow r_{total} + \gamma^{(t_{total}+t)} r/(1 - \gamma^t)$
17:            $t_{total} \leftarrow \infty$
18:        else if the trajectory stopped because the final state was absorbing then
19:            $t_{total} \leftarrow \infty$
20:        end if
21:        $s_{stop} \leftarrow s$
22:        Add $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total} \rangle$ to the trajectory list in $p$
23:    end for
24: end for
25: Add $p$ to $l$
```

Procedure 10: Procedure $\mathrm{UpdateAbstractSMDP}(\mathit{tree}, \bar{\mathcal{A}}, G, \gamma)$

```
1: **for all** leaves $l$ with fewer than $N_l$ sample points **do**
2:     $\mathcal{S}_a \leftarrow \{s_1, \ldots, s_{N_a}\}$ sampled from $l$
3:     **for all** $s \in \mathcal{S}_a$ **do**
4:         $\mathrm{SampleTrajectories}(s, \mathit{tree}, \bar{\mathcal{A}}, G, \gamma)$ {see Procedure 9}
5:     **end for**
6: **end for**
7: $\mathcal{P} \leftarrow \emptyset$ {Reset abstract transition count}
8: **for all** leaves $l$ and associated points $p$ **do**
9:     **for all** trajectories, $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total}\rangle$, in $p$ **do**
10:        $l_{stop} \leftarrow \mathrm{LeafContaining}(\mathit{tree}, s_{stop})$
11:        $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle l, \bar{a}, l_{stop}, t_{total}, r_{total}\rangle\}$
12:    **end for**
13: **end for**
14: Transform $\mathcal{P}$ into transition probabilities
15: Solve the abstract SMDP
```

As noted above (equation 5.9 – reprinted here), for a single trajectory, $\langle s_{start}, \bar{a_j}, s_{stop}, r, t\rangle$, we can find $\bar{s}_{stop}$ and can calculate $\hat{T}$:

$$\hat{T}_{\bar{s}_{start}}(s_{start}, \bar{a}) = r + \gamma^t \bar{V}(\bar{s}_{stop}) \tag{5.10}$$

For a set of trajectories all starting at the same base level state with the same abstract action we find a better estimate:

$$\hat{T}_{\bar{s}_{start}}(s_{start}, \bar{a}) = \frac{1}{N_t} \sum_{i=0}^{N_t} \left[r_i + \gamma^{t_i} \bar{V}(\bar{s}_{stop_i}) \right] \tag{5.11}$$

This is the estimated expected discounted reward for following the abstract action $\bar{a}$ starting at the base level state s_{start}, until a new abstract state is reached, and then following the policy defined by the abstract SMDP. If there is a statistically significant change in the $\hat{T}$ value across a state for any action then we should divide the state in two.

Additionally, we can find which abstract action has the highest $\hat{T}$ estimate[2], $\hat{\pi}$, and the value of that estimate, $\hat{V}$:

[2]Ties are broken in favor of the abstract action selected in the current abstract state.

$$\hat{V}(s_{start}) = \max_{\bar{a}} \hat{T}(s_{start}, \bar{a}) \tag{5.12}$$

$$\hat{\pi}(s_{start}) = \operatorname*{argmax}_{\bar{a}} \hat{T}(s_{start}, \bar{a}) \tag{5.13}$$

If the $\hat{V}$ or $\hat{\pi}$ value changes across an abstract state, then we should divide that abstract state in two. Note that it is impossible for $\hat{\pi}(s)$ or $\hat{V}(s)$ to change without $\hat{T}(s, \bar{a})$ changing and so these extra criteria do not cause us to introduce any extra splits. However, they do change the order in which splits are introduced. Splits that would allow a change in policy, or a change in value function, are preferred over those that just improve our estimate of the Q function.

In Chapter 3 we used a non-parametric test, the Kolmogorov-Smirnov test, to decide when to split an abstract state. The experiments in this chapter use a Minimum Description Length (MDL) based test, similar to the one in Chapter 4. The division that maximizes the statistical difference between the two sides is chosen.

There are two differences between the MDL test in Lumberjack and the one that is used by TTree. Both differences are in the encoding of the values in the leaves. Firstly, the TTree encoding is slightly redundant in that $\hat{V}$, $\hat{\pi}$ and $\hat{T}$ values are all encoded in the leaves. The $\hat{\pi}$ values are categorical and can be encoded exactly as in Lumberjack. The $\hat{V}$ and $\hat{T}$ values are continuous and need to be encoded differently.

These continuous values are encoded as follows. Firstly, the means, μ_V and $\{\mu_{\bar{a}0}, \ldots, \mu_{\bar{a}n}\}$, and standard deviations, σ_V and $\{\sigma_{\bar{a}0}, \ldots, \sigma_{\bar{a}n}\}$, of the continuous values in each leaf are recorded, each using $-\log_2(C_{const})$ bits, where $C_{const} = 10^{-10}$ is a constant representing the accuracy with which we wish to encode these values. Then the probability of each value under that normal distribution is found, $p_i^V = N_{\mu_V,\sigma_V}\left(\hat{V}(s_i)\right)$, $p_i^{\bar{a}} = N_{\mu_{\bar{a}},\sigma_{\bar{a}}}\left(\hat{T}(s_i, \bar{a})\right)$. Finally, each value is encoded using $-\log_2(C_{const}\ p)$ bits. This encoding is closely related to the sum-squared error test used in Continuous U Tree.

We now have the code lengths of three different sets of values: $C(\hat{V})$, $C(\hat{\pi})$ and $C(\hat{T})$. These are combined into a code length for the leaf as follows:

$$C_{leaf} = C(\hat{V}) + 20C(\hat{\pi}) \tag{5.14}$$

If the C_{leaf} values don't introduce a split, then the $C(\hat{T})$ values are added. The result is that all splits based on value function and policy are introduced before splits based on non-optimal actions.

Procedure 11: Procedure $\mathrm{GrowTTree}(tree, \bar{\mathcal{A}}, \gamma)$

```
1: $\mathcal{D}^T \leftarrow \emptyset$ {Reset split data set. $\mathcal{D}^T$ is a set of states with associated $\hat{T}$ estimates.}
2: $\mathcal{D}^\pi \leftarrow \emptyset$
3: $\mathcal{D}^V \leftarrow \emptyset$
4: **for all** leaves $l$ and associated points $p$ **do** {a point contains a set of trajectories starting in the same state}
5:     $\hat{T}(s_{start}, .) \leftarrow \emptyset$ {$\hat{T}(s_{start}, .)$ is a new array of size $|\bar{\mathcal{A}}|$}
6:     **for all** trajectories in $p$, $\langle s_{start}, \bar{a}, s_{stop}, t, r\rangle$ **do** {$N_t$ trajectories for each of $|\bar{\mathcal{A}}|$ actions}
7:         $l_{stop} \leftarrow \mathrm{LeafContaining}(tree, s_{stop})$
8:         $\hat{T}(s_{start}, \bar{a}) \leftarrow \hat{T}(s_{start}, \bar{a}) + (r + \gamma^t V(l_{stop}))/N_t$
9:     **end for**
10:    $\mathcal{D}^T \leftarrow \mathcal{D}^T \cup \{\langle s_{start}, \hat{T}\rangle\}$ {add $\hat{T}$ estimates to data set}
11:    $\hat{V} \leftarrow \max_{\bar{a}} \hat{T}(s_{start}, \bar{a})$
12:    $\hat{\pi} \leftarrow \mathrm{argmax}_{\bar{a}} \hat{T}(s_{start}, \bar{a})$
13:    $\mathcal{D}^V \leftarrow \mathcal{D}^V \cup \{\langle s, \hat{V}\rangle\}$ {add best value to data set}
14:    $\mathcal{D}^\pi \leftarrow \mathcal{D}^\pi \cup \{\langle s, \hat{\pi}\rangle\}$ {add best action to data set}
15: **end for**
16: **for all** new splits in the tree **do**
17:     $\mathrm{EvaluateSplit}(\mathcal{D}^V \cup \mathcal{D}^\pi \cup \mathcal{D}^T)$ {Use the splitting criterion to evaluate this split }
18: **end for**
19: **if** $\mathrm{ShouldSplit}(\mathcal{D}^V \cup \mathcal{D}^\pi \cup \mathcal{D}^T)$ **then** {Evaluate the best split using the stopping criterion}
20:     Introduce best split into tree
21:     Throw out all sample points, $p$, in the leaf that was split
22: **end if**
```

As well as knowing *how* to grow a tree, we also need to decide *if* we should grow the tree. This is decided by a stopping criterion. Procedure 11 does not introduce a split if the stopping criterion is fulfilled, but neither does it halt the algorithm. TTree keeps looping gathering more data. In the experimental results we use a Minimum Description Length stopping criterion. We have found that the algorithm tends to get very good results long before the stopping criterion is met, and we did not usually run the algorithm for that long. The outer loop in Procedure 8 is an infinite loop, although it is possible to modify the algorithm so that it stops when the stopping criterion is fulfilled. We have been using the algorithm as an anytime algorithm.

5.2.4 Discussion of TTree

Now that we have described the technical details of the algorithm, we look at the motivation and effects of these details. TTree was developed to fix some of the limitations of Continuous U Tree described in Section 3.5. In particular we wanted to reduce the splitting from the edges of abstract states and we wanted to allow the measurement of the usefulness of abstract actions. Finally, we wanted to improve the match between the way the abstract policy is used and the way the abstract SMDP is modelled to increase the quality of the policy when the tree is not fully grown.

Introducing trajectories instead of transitions solves these problems. The $\hat{T}$ values, unlike the q values in Continuous U Tree, vary all across an abstract state, solving the edge slicing issue. The use of trajectories allows us to measure the effectiveness of abstract actions along a whole trajectory rather than only for a single step. Finally, the use of trajectories allows us to build a more accurate abstract transition function.

Edge slicing was an issue in Continuous U Tree where all abstract self-transitions with the same reward had the same q values, regardless of the dynamics of the self-transition. This means that often only the transitions out of an abstract state have different q values, and hence that the algorithm tends to slice from the edges of abstract states into the middle. TTree does not suffer from this problem as the trajectories include a measure of how much time the agent spends following the trajectory before leaving the abstract state. If the state-dynamics change across a state, then that is apparent in the $\hat{T}$ values.

Trajectories allow us to select abstract actions for a state because they provide a way to differentiate abstract actions from base level actions. In one step there is no way to differentiate an abstract action from a base level action. Over multiple steps, this becomes possible.

Finally, trajectories allow a more accurate transition function because they more accurately model the execution of the abstract policy. When the abstract SMDP is solved, an abstract action is selected for each abstract state. During execution that action is executed repeatedly until the agent leaves the abstract state. This repeated execution until the abstract state is exited is modelled by a trajectory. This is different from how Continuous U Tree forms its abstract MDP where each step is modelled individually. TTree only applies the Markov assumption at the start of a trajectory, whereas Continuous U Tree applies it at each step. When the tree is not fully grown, and the Markov assumption inaccurate, fewer applications of the assumption lead to a more accurate model.

However, the use of trajectories also brings its own issues. If the same action is always selected until a new abstract state is reached, then we have lost the ability to change direction halfway across an abstract state. Our first answer to this is to sample trajectories from random starting points throughout the state, as described above. This allows us to measure the effect of changing direction in a state by starting a new trajectory in that state. To achieve this we require a generative model of the world. With this sampling, if the optimal policy changes halfway across a state, then the $\hat{T}$ values should change. But we only get $\hat{T}$ values where we start trajectories.

It is not clear that we can find the optimal policy in this constrained model. In fact, with a fixed size tree we usually can not find the optimal policy, and hence we need to grow the tree. With a large enough tree the abstract states and base level states are equivalent, so we know that expanding the tree can lead to optimality. However, it is still not obvious that the $\hat{T}$ values contain the information we need to decide if we should keep expanding the tree; *i.e.* it is not obvious that there are no local maxima, with all the $\hat{T}$ values equal within all leaves, but with a non-optimal policy. We prove that no such local maxima exist in Section 5.3 below.

The fact that we split first on $\hat{V} = \max_{\bar{a}} \hat{T}(., \bar{a})$ and $\hat{\pi} = \operatorname{argmax}_{\bar{a}} \hat{T}(., \bar{a})$ values before looking at all the $\hat{T}$ values deserves some explanation. If you split on $\hat{T}$ values then you sometimes split based on the data for non-optimal abstract actions. While this is required for the proof in Section 5.3 (see the example in Section 5.3.2), it also tends to cause problems empirically (see the example in Section 5.5). Our solution is to only split on non-optimal actions when no splits would otherwise be introduced.

Finally, we make some comments about the random abstract action. The random abstract action has $\hat{T}$ values that are a smoothed version of the reward function. We saw with

Continuous U Tree that there could be a problem finding an initial split if there was a single point reward. The point reward may not be sampled often enough to find a statistically significant difference between it and surrounding states. The random abstract action improves the chance of finding the point reward and introducing the initial split. In some of the empirical results we generalize this to the notion of an abstract action for exploration.

5.3 Proof of Convergence

Previous state abstraction algorithms, like Continuous U Tree described in Chapter 3, have generated data in a manner similar to TTree, but using single transitions rather than trajectories. In that case, the data can be interpreted as a sample from a stochastic form of the Q-function (TTree exhibits this behaviour as a special case when MAXTIME $= 0$). When trajectories are introduced, the sample values no longer have this interpretation and it is no longer clear that splitting on the sample values leads to an abstract SMDP with any formal relation to the original SMDP.

In this section, we analyze the trajectory values and show that splitting so that the $\hat{T}$ values are equal for all actions across a leaf leads to the optimal policy for the abstract SMDP, $\bar{\pi}^*$, also being an optimal policy for the original SMDP. We also give a counter-example for a simplified version of TTree showing that having constant trajectory values for only the highest valued action is not enough to achieve optimality.

5.3.1 Assumptions

In order to separate the effectiveness of the splitting and stopping criteria from the convergence of the SMDP solving, we assume optimal splitting and stopping criteria and that the sample sizes, N_l and N_t, are sufficient. That is, the splitting and stopping criteria introduce a split in a leaf if, and only if, there exist two regions, one on each side of the split, and the distribution of the value being tested is different in those regions.

Of course, real world splitting criteria are not optimal, even with infinite sample sizes. For example, most splitting criteria have trouble introducing splits if the data follows an XOR or checkerboard pattern. Our assumption is still useful as it allows us to verify the correctness of the SMDP solving part of the algorithm independently of the splitting and stopping criteria.

This proof only refers to base level actions. We assume that the only abstract actions are the automatically generated, degenerate abstract actions, and hence $\forall \bar{a}, \forall s, \pi_{\bar{a}}(s) = a$ and we do not have to distinguish between a and $\bar{a}$. Adding extra abstract actions does not affect the proof, and so we ignore them for convenience of notation.

Theorem 1 *If the $\hat{T}$ samples are statistically constant across all states for all actions, then an optimal abstract policy is an optimal base level policy. Formally,*

$$\forall \bar{a} \in \bar{\mathcal{A}}, \forall \bar{s} \in \bar{\mathcal{S}}, \forall s_1 \in \bar{s}, \forall s_2 \in \bar{s}, \tilde{T}(s_1, \bar{a}) = \tilde{T}(s_2, \bar{a}) \Rightarrow \bar{\pi}^*(s_1) = \pi^*(s_1) \tag{5.15}$$

We first review the definition of $\tilde{T}$ introduced in Equation 5.4:

$$\begin{aligned} \tilde{T}_{\bar{s}}(s,a) = R(s,a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{T}_{\bar{s}}(s',a)\,\mathrm{d}t \\ &+ \sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t \end{aligned} \tag{5.16}$$

This function is the expected value of the $\hat{T}$ samples used in the algorithm, assuming a large sample size. It is also closely related to the T function defined in equation 5.3; the two are identical except for the value used when the region defined by $\mathcal{S}'$ or $\bar{s}$ is exited. The T function used the value of a base level value function, V, whereas the $\tilde{T}$ function uses the value of the abstract level value function, $\bar{V}$.

We also define functions $\tilde{V}_{\bar{s}}^*(s)$ and $\tilde{Q}_{\bar{s}}^*(s,a)$ to be similar to the normal V^* and Q^* functions within the set of states corresponding to $\bar{s}$, but once the agent leaves $\bar{s}$ it gets a one-time reward equal to the value of the abstract state it enters, $\bar{V}$.

$$\begin{aligned} \tilde{Q}_{\bar{s}}^*(s,a) = R(s,a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{V}_{\bar{s}}^*(s')\,\mathrm{d}t \\ &+ \sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t \end{aligned} \tag{5.17}$$

$$\tilde{V}_{\bar{s}}^*(s) = \max_a \tilde{Q}_{\bar{s}}^*(s,a) \tag{5.18}$$

Intuitively these functions give the value of acting optimally within $\bar{s}$, assuming that the values of the base level states outside $\bar{s}$ are fixed.

We now have a spectrum of functions. At one end of the spectrum is the base Q^* function from which it is possible to extract the set of optimal policies for the original SMDP. Next in line is the $\tilde{Q}^*$ function which is optimal within an abstract state given the values of the abstract states around it. Then we have the $\tilde{T}$ function which can have different values across an abstract state, but assumes a constant action until a new abstract state is reached. Finally we have the abstract $\bar{Q}^*$ function which does not vary across the abstract state and gives us the optimal policy for the abstract SMDP.

The outline of the proof of optimality when splitting is complete is as follows. First, we show in Lemma 1 that $\tilde{T}$ really is the same as our estimates, $\hat{T}$, for large enough sample sizes. We then show that, when splitting has stopped, the maximum over the actions of each of the functions in the spectrum mentioned in the previous paragraph is equal and is reached by the same set of actions. We also show that $\bar{Q}^* \leq Q^*$. This implies that an optimal policy in the abstract SMDP is also an optimal policy in the base SMDP.

Lemma 1 *The $\hat{T}$ samples are an unbiased estimate of $\tilde{T}$. Formally,*

$$\underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{E} \hat{T}_{\bar{s}}(s,\bar{a}) = \tilde{T}_{\bar{s}}(s,a) \tag{5.19}$$

Consider $\underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{\mathrm{E}} \hat{T}_{\bar{s}}(s,\bar{a})$:

$$\underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{\mathrm{E}} \hat{T}_{\bar{s}}(s,\bar{a}) \tag{5.20}$$

$$= \underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{\mathrm{E}} \left[r + \gamma^t \bar{V}(\bar{s'})\right] \qquad \text{by substituting in equation 5.9} \tag{5.21}$$

$$= \underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{\mathrm{E}} r + \underset{\substack{\text{trajectories starting at } s \\ \langle s,\bar{a},s',r,t\rangle}}{\mathrm{E}} \gamma^t \bar{V}(\bar{s'}) \tag{5.22}$$

We now move from samples to the $\tilde{R}$ and $\tilde{P}$ functions defined in equations 5.5 and 5.6. Assuming that sample sizes are large, these values are equivalent.

$$= \tilde{R}_{\bar{s}}(s,a) + \sum_{\bar{s'} \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s'},t) \gamma^t \bar{V}(\bar{s'})\, \mathrm{d}t \tag{5.23}$$

We then separate the current abstract state from the rest.

$$\begin{aligned}=&\tilde{R}_{\bar{s}}(s,a)+\int_{t=0}^{\infty}\tilde{P}_{s,a}(\bar{s},t)\gamma^t\bar{V}(\bar{s})\,\mathrm{d}t+\\&\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\tilde{P}_{s,a}(\bar{s'},t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned}\tag{5.24}$$

But $\tilde{P}_{s,.}(\bar{s},t)$ is only non-zero when $t=\infty$. This is multiplied by γ^t and hence the self-transition disappears, as noted in the commentary when $\tilde{P}$ was defined (immediately above Equation 5.6).

$$=\tilde{R}_{\bar{s}}(s,a)+\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\tilde{P}_{s,a}(\bar{s'},t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\tag{5.25}$$

Now we substitute in the definitions of $\tilde{R}$ and $\tilde{P}$ from equations 5.5 and 5.6.

$$\begin{aligned}=&\left[R(s,a)+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t\right]+\\&\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\left[\sum_{s''\in\bar{s'}}P_{s,a}(s'',t)+\right.\\&\left.\sum_{s''\in\bar{s}}\int_{t'=0}^{t}P_{s,a}(s'',t')\tilde{P}_{s'',a}(\bar{s'},t-t')\,\mathrm{d}t'\right]\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned}\tag{5.26}$$

$$\begin{aligned}=&R(s,a)+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t+\\&\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\sum_{s''\in\bar{s'}}P_{s,a}(s'',t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t+\\&\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\sum_{s''\in\bar{s}}\int_{t'=0}^{t}P_{s,a}(s'',t')\tilde{P}_{s'',a}(\bar{s'},t-t')\,\mathrm{d}t'\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned}\tag{5.27}$$

Then we rewrite the pair of summations on the second line as a single summation.

$$\begin{aligned}=&R(s,a)+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t+\\&\sum_{s''\in(\mathcal{S}-\bar{s})}\int_{t=0}^{\infty}P_{s,a}(s'',t)\gamma^t\bar{V}(\bar{s''})\,\mathrm{d}t+\\&\sum_{\bar{s'}\in(\bar{\mathcal{S}}-\{\bar{s}\})}\int_{t=0}^{\infty}\sum_{s''\in\bar{s}}\int_{t'=0}^{t}P_{s,a}(s'',t')\tilde{P}_{s'',a}(\bar{s'},t-t')\,\mathrm{d}t'\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned}\tag{5.28}$$

We reverse the order of the second and third lines.

$$\begin{aligned}
=&R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{\bar{s'} \in (\bar{\mathcal{S}} - \{\bar{s}\})} \int_{t=0}^{\infty} \sum_{s'' \in \bar{s}} \int_{t'=0}^{t} P_{s,a}(s'',t')\tilde{P}_{s'',a}(\bar{s'}, t-t')\,\mathrm{d}t' \gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t + \\
&\sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned} \tag{5.29}$$

We swap s' and s'' in the second line. This is simply a variable rename. Also, we move together the two integrals in the second line.

$$\begin{aligned}
=&R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{\bar{s''} \in (\bar{\mathcal{S}} - \{\bar{s}\})} \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} \int_{t'=0}^{t} P_{s,a}(s',t')\tilde{P}_{s',a}(\bar{s''}, t-t')\gamma^t \bar{V}(\bar{s''})\,\mathrm{d}t'\,\mathrm{d}t + \\
&\sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned} \tag{5.30}$$

We replace $\int_{t=0}^{\infty} \int_{t'=0}^{t} X\,\mathrm{d}t'\,\mathrm{d}t$ with $\int_{t'=0}^{\infty} \int_{t=t'}^{\infty} X\,\mathrm{d}t\,\mathrm{d}t'$.

$$\begin{aligned}
=&R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{\bar{s''} \in (\bar{\mathcal{S}} - \{\bar{s}\})} \sum_{s' \in \bar{s}} \int_{t'=0}^{\infty} \int_{t=t'}^{\infty} P_{s,a}(s',t')\tilde{P}_{s',a}(\bar{s''}, t-t')\gamma^t \bar{V}(\bar{s''})\,\mathrm{d}t\,\mathrm{d}t' + \\
&\sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned} \tag{5.31}$$

We split γ^t into $\gamma^{t'}\gamma^{(t-t')}$ in the second line.

$$\begin{aligned}
=&R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{\bar{s''} \in (\bar{\mathcal{S}} - \{\bar{s}\})} \sum_{s' \in \bar{s}} \int_{t'=0}^{\infty} \int_{t=t'}^{\infty} P_{s,a}(s',t')\tilde{P}_{s',a}(\bar{s''}, t-t')\gamma^{t'}\gamma^{(t-t')} \bar{V}(\bar{s''})\,\mathrm{d}t\,\mathrm{d}t' + \\
&\sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned} \tag{5.32}$$

We re-order the summations on the second line and bring $P_{s,a}(s',t')\gamma^{t'}$ outside some of the summations.

$$
\begin{aligned}
=&R(s,a) + \sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{s'\in\bar{s}} \int_{t'=0}^{\infty} P_{s,a}(s',t')\gamma^{t'} \sum_{\bar{s''}\in(\bar{\mathcal{S}}-\{\bar{s}\})} \int_{t=t'}^{\infty} \tilde{P}_{s',a}(\bar{s''},t-t')\gamma^{(t-t')}\bar{V}(\bar{s''})\,\mathrm{d}t\,\mathrm{d}t' + \\
&\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned}
\tag{5.33}
$$

We swap t and t' variable names on the second line. This is just a change in variable names.

$$
\begin{aligned}
=&R(s,a) + \sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{R}_{\bar{s}}(s',a)\,\mathrm{d}t + \\
&\sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^{t} \sum_{\bar{s''}\in(\bar{\mathcal{S}}-\{\bar{s}\})} \int_{t'=t}^{\infty} \tilde{P}_{s',a}(\bar{s''},t'-t)\gamma^{(t'-t)}\bar{V}(\bar{s''})\,\mathrm{d}t'\,\mathrm{d}t + \\
&\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned}
\tag{5.34}
$$

We re-order the summations again to bring $\sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t$ out of the first and second lines.

$$
\begin{aligned}
=&R(s,a) + \sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \Big[\tilde{R}_{\bar{s}}(s',a) + \\
&\sum_{\bar{s''}\in(\bar{\mathcal{S}}-\{\bar{s}\})} \int_{t'=t}^{\infty} \tilde{P}_{s',a}(\bar{s''},t'-t)\gamma^{(t'-t)}\bar{V}(\bar{s''})\,\mathrm{d}t'\Big]\,\mathrm{d}t + \\
&\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned}
\tag{5.35}
$$

We subtract t from t' in the integral in the second line, then add it back to each reference to t' and cancel the terms.

$$
\begin{aligned}
=&R(s,a) + \sum_{s'\in\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \Big[\tilde{R}_{\bar{s}}(s',a) + \\
&\sum_{\bar{s''}\in(\bar{\mathcal{S}}-\{\bar{s}\})} \int_{t'=0}^{\infty} \tilde{P}_{s',a}(\bar{s''},t')\gamma^{t'}\bar{V}(\bar{s''})\,\mathrm{d}t'\Big]\,\mathrm{d}t + \\
&\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t
\end{aligned}
\tag{5.36}
$$

The section of Equation 5.36 in square brackets matches Equation 5.23. We substitute Equation 5.20, which is equivalent.

$$\begin{aligned}=&R(s,a)+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\,\mathrm{E}\,\hat{T}_{\bar{s}}(s',a)\,\mathrm{d}t+\\ &\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned} \tag{5.37}$$

The above line is equal to Equation 5.20. We have a recursive function. This recursive function is of the same form as Equation 5.4. Any solution to one recursive definition must be a solution to the other also. Hence,

$$=\tilde{T}_{\bar{s}}(s,a) \tag{5.38}$$

□

Lemma 2 $\forall s\in\bar{s},\forall a,\tilde{Q}_{\bar{s}}^*(s,a)\geq\tilde{T}_{\bar{s}}(s,a)$

This is true by inspection. Equations 5.4 and 5.17 are reprinted here for reference:

$$\begin{aligned}\tilde{T}_{\bar{s}}(s,a)=&R(s,a)\\ &+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\tilde{T}_{\bar{s}}(s',a)\,\mathrm{d}t\\ &+\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned} \tag{5.39}$$

$$\begin{aligned}\tilde{Q}_{\bar{s}}^*(s,a)=&R(s,a)\\ &+\sum_{s'\in\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\tilde{V}_{\bar{s}}^*(s')\,\mathrm{d}t\\ &+\sum_{s'\in\bar{s'},\bar{s'}\neq\bar{s}}\int_{t=0}^{\infty}P_{s,a}(s',t)\gamma^t\bar{V}(\bar{s'})\,\mathrm{d}t\end{aligned} \tag{5.40}$$

Substituting $\tilde{V}_{\bar{s}}^*(s)=\max_a\tilde{Q}_{\bar{s}}^*(s,a)$ into Equation 5.40 makes the two functions differ only in that $\tilde{Q}$ has a $\max$ where $\tilde{T}$ does not. Hence $\tilde{Q}\geq\tilde{T}$.

□

Lemma 3 *If $\tilde{T}_{\bar{s}}$ is constant across $\bar{s}$ for all actions, then* $\max_a\tilde{T}_{\bar{s}}(.,a)=\max_a\tilde{Q}_{\bar{s}}^*(.,a)$ *and* $\operatorname{argmax}_a\tilde{T}_{\bar{s}}(.,a)=\operatorname{argmax}_a\tilde{Q}_{\bar{s}}^*(.,a)$.

Consider $\tilde{Q}_{\bar{s}}^{*}(s,a) - \tilde{T}_{\bar{s}}(s,a)$:

$$\begin{aligned}\tilde{Q}_{\bar{s}}^{*}(s,a) - \tilde{T}_{\bar{s}}(s,a) = &\Bigg(R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{V}_{\bar{s}}^{*}(s')\,\mathrm{d}t \\ &\quad + \sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t\Bigg) \\ &- \Bigg(R(s,a) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{T}_{\bar{s}}(s',a)\,\mathrm{d}t \\ &\quad + \sum_{s' \in \bar{s'}, \bar{s'} \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \bar{V}(\bar{s'})\,\mathrm{d}t\Bigg)\end{aligned} \tag{5.41}$$

$$\begin{aligned}= &\sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{V}_{\bar{s}}^{*}(s')\,\mathrm{d}t \\ &- \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \tilde{T}_{\bar{s}}(s',a)\,\mathrm{d}t\end{aligned} \tag{5.42}$$

$$= \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s',t)\gamma^t \left[\tilde{V}_{\bar{s}}^{*}(s') - \tilde{T}_{\bar{s}}(s',a)\right]\,\mathrm{d}t \tag{5.43}$$

Consider $\tilde{Q}_{\bar{s}}^{*}(s,a) - \tilde{T}_{\bar{s}}(s,a)$ in the state/action pair in $\bar{s}$ with the largest value of $\tilde{Q}_{\bar{s}}^{*}$, s^*, a^*. By assumption, $\forall s, s' \in \bar{s}, \forall a, \tilde{T}_{\bar{s}}(s,a) = \tilde{T}_{\bar{s}}(s',a)$. So,

$$\begin{aligned}\tilde{Q}_{\bar{s}}^{*}(s^*,a^*) - \tilde{T}_{\bar{s}}(s^*,a^*) &\geq \max_a \tilde{Q}_{\bar{s}}^{*}(s',a) - \tilde{T}_{\bar{s}}(s^*,a^*) \qquad \text{for any } s' \in \bar{s} \\ &= \max_a \tilde{Q}_{\bar{s}}^{*}(s',a) - \tilde{T}_{\bar{s}}(s',a^*) \\ &= \tilde{V}_{\bar{s}}^{*}(s') - \tilde{T}_{\bar{s}}(s',a^*)\end{aligned} \tag{5.44}$$

The sum and integral in equation 5.43 are simply taking an expectation, and there is a factor of γ giving:

$$\begin{aligned}&\sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s^*,a^*}(s',t)\gamma^t \left[\tilde{V}_{\bar{s}}^{*}(s') - \tilde{T}_{\bar{s}}(s',a^*)\right]\,\mathrm{d}t \\ &\qquad \leq \gamma^{\alpha}\left[\tilde{Q}_{\bar{s}}^{*}(s^*,a^*) - \tilde{T}_{\bar{s}}(s^*,a^*)\right] \qquad \text{where } \alpha > 0\end{aligned} \tag{5.45}$$

Hence:

$$\tilde{Q}_{\bar{s}}^{*}(s^*, a^*) - \tilde{T}_{\bar{s}}(s^*, a^*) \leq \gamma^{\alpha} \left[\tilde{Q}_{\bar{s}}^{*}(s^*, a^*) - \tilde{T}_{\bar{s}}(s^*, a^*)\right] \tag{5.46}$$

$$\therefore \tilde{Q}_{\bar{s}}^{*}(s^*, a^*) - \tilde{T}_{\bar{s}}(s^*, a^*) = 0 \tag{5.47}$$

$$\therefore \tilde{Q}_{\bar{s}}^{*}(s^*, a^*) = \tilde{T}_{\bar{s}}(s^*, a^*) \tag{5.48}$$

But, $\tilde{T}_{\bar{s}}(s, a)$ is constant across all $s \in \bar{s}$ for all a. Moreover, as shown above in lemma 2, $\tilde{Q} \geq \tilde{T}$. Hence:

$$\forall s' \in \mathcal{S}', \tilde{T}_{\bar{s}}(s', a^*) = \tilde{Q}_{\bar{s}}^{*}(s, a^*) \tag{5.49}$$

□

Lemma 4 *If $\tilde{T}_{\bar{s}}$ is constant across the abstract state $\bar{s}$ for all actions then $\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a)$ for all actions.*

During the proof of lemma 1 we saw that (equation 5.23),

$$\tilde{T}_{\bar{s}}(s, a) = \tilde{R}_{\bar{s}}(s, a) + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s}', t) \gamma^t \bar{V}(\bar{s}') \, \mathrm{d}t$$

Now,

$$\bar{Q}(\bar{s}, a) = \bar{R}(\bar{s}, a) + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \bar{P}_{\bar{s},a}(\bar{s}', t) \gamma^t \bar{V}^{\pi}(\bar{s}') \, \mathrm{d}t \tag{5.50}$$

Substituting equations 5.7 and 5.8,

$$= \mathop{\mathrm{E}}_{s\in\bar{s}}\left[\tilde{R}_{\bar{s}}(s,a)\right] + \sum_{\bar{s'}\in\bar{\mathcal{S}}}\int_{t=0}^{\infty}\mathop{\mathrm{E}}_{s\in\bar{s}}\left[\tilde{P}_{s,a}(\bar{s'},t)\right]\gamma^t\bar{V}^{\pi}(\bar{s'})\,\mathrm{d}t \tag{5.51}$$

$$\begin{aligned} &= \frac{1}{|\bar{s}|}\left[\sum_{s\in\bar{s}}\tilde{R}_{\bar{s}}(s,a)\right] \\ &\quad + \sum_{\bar{s'}\in\bar{\mathcal{S}}}\int_{t=0}^{\infty}\frac{1}{|\bar{s}|}\left[\sum_{s\in\bar{s}}\tilde{P}_{s,a}(\bar{s'},t)\right]\gamma^t\bar{V}^{\pi}(\bar{s'})\,\mathrm{d}t \end{aligned} \tag{5.52}$$

$$\begin{aligned} &= \frac{1}{|\bar{s}|}\sum_{s\in\bar{s}}\tilde{R}_{\bar{s}}(s,a) \\ &\quad + \frac{1}{|\bar{s}|}\sum_{s\in\bar{s}}\sum_{\bar{s'}\in\bar{\mathcal{S}}}\int_{t=0}^{\infty}\tilde{P}_{s,a}(\bar{s'},t)\gamma^t\bar{V}^{\pi}(\bar{s'})\,\mathrm{d}t \end{aligned} \tag{5.53}$$

$$\begin{aligned} &= \frac{1}{|\bar{s}|}\sum_{s\in\bar{s}}\left[\tilde{R}_{\bar{s}}(s,a)\right. \\ &\quad \left. + \sum_{\bar{s'}\in\bar{\mathcal{S}}}\int_{t=0}^{\infty}\tilde{P}_{s,a}(\bar{s'},t)\gamma^t\bar{V}^{\pi}(\bar{s'})\,\mathrm{d}t\right] \end{aligned} \tag{5.54}$$

$$= \mathop{\mathrm{E}}_{s\in\bar{s}}\tilde{T}_{\bar{s}}(s,a) \tag{5.55}$$

But given that $\tilde{T}_{\bar{s}}(s,a)$ is constant across $s\in\bar{s}$, we know that $\forall s'\in\bar{s}, \mathrm{E}_{s\in\bar{s}}\tilde{T}_{\bar{s}}(s,a) = \tilde{T}_{\bar{s}}(s',a)$.

□

Lemma 5 *If $\tilde{T}_{\bar{s}}$ is constant across the abstract state $\bar{s}$ for all actions, and $\bar{V}(\bar{s'}) = V^*(s')$ for all base level states in all abstract states $\bar{s'}$, $\bar{s'}\neq\bar{s}$, then $\bar{V}(\bar{s}) = V^*(s)$ in $\bar{s}$.*

Substituting $\bar{V}(\bar{s'}) = V^*(s')$ for other states into equation 5.17, we see that $\tilde{Q}^* = Q^*$ for the current state and so $\tilde{V}^* = V^*$ for the current state. Also, $\mathrm{argmax}_a\,\tilde{Q}^*(s,a) = \mathrm{argmax}_a\,Q^*(s,a)$ and so the policies implicit in these functions are also equal.

Moreover, because $\tilde{T}_{\bar{s}}$ is constant across the current abstract state, we know, by Lemma 4, that $\bar{Q}(\bar{s},a) = \tilde{T}_{\bar{s}}(s,a)$. For the same reason we also know by Lemma 3 that $\max_a \tilde{T}_{\bar{s}}(s,a) = \tilde{V}^*_{\bar{s}}(s)$.

So,

$$\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a) \tag{5.56}$$
$$\therefore \bar{V}(\bar{s}) = \max_a \tilde{T}_{\bar{s}}(s, a) \tag{5.57}$$
$$= \tilde{V}_{\bar{s}}^*(s) \tag{5.58}$$
$$= V^*(s) \tag{5.59}$$

□

Lemma 6 *If $\tilde{T}_{\bar{s}}$ is constant across each abstract state for each action, then setting $V^* = \bar{V}$ is a consistent solution to the Bellman equations of the base level SMDP.*

This is most easily seen by contradiction. Assume we have a tabular representation of the base level value function. We will initialize this table with the values from $\bar{V}$. We will further assume that $\tilde{T}_{\bar{s}}$ is constant across each abstract state for each action, but that our table is not optimal, and show that this leads to a contradiction.

As in lemma 5, because $\tilde{T}_{\bar{s}}$ is constant across the current abstract state, we know, by Lemma 4, that $\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a)$. For the same reason we also know by Lemma 3 that $\max_a \tilde{T}_{\bar{s}}(s, a) = \tilde{V}_{\bar{s}}^*(s)$.

This means that our table contains $\tilde{V}_{\bar{s}}^*$ for each abstract state. Hence, there is no single base level state that can have its value increased by a single bellman update. Hence the table must be optimal.

This optimal value function is achieved with the same actions in both the base and abstract SMDPs. Hence any optimal policy in one is an optimal policy in the other.

□

5.3.2 Splitting on non-optimal actions

We did not show above that the $\tilde{T}$ and Q^* functions are equal for non-optimal actions. One might propose a simpler algorithm that only divides a state when $\tilde{T}$ is not uniform for the action with the highest value, rather than checking for uniformity all the actions. Here is a counter-example showing this simplified algorithm does not converge.

Consider an MDP with three states, s_1, s_2 and s_3. s_3 is an absorbing state with zero value. States s_1 and s_2 are both part of a single abstract state, s_3 is in a separate abstract state. There are two deterministic actions. a_1 takes us from either state into s_3 with a reward of 10. a_2 takes us from s_1 to s_2 with a reward of 100, and from s_2 to s_3 with a reward of -1000. Table 5.2 shows the $\tilde{T}$ and Q^* values for each state when $\gamma = 0.9$. Note that even though the $T(s, a_1)$ values are constant and higher than the $T(s, a_2)$ values, the optimal policy does not choose action a_1 in both states.

Function	s_1	s_2
$Q(s, a_1)$	9	9
$T(s, a_1)$	9	9
$Q(s, a_2)$	108.1	-900
$T(s, a_2)$	-710	-900
$V(s)$	108.1	9

Table 5.2: T, Q and V for sample MDP.

5.4 An Example TTree Execution

This example describes TTree running in the Towers of Hanoi domain. The full domain is described in Section 2.6.3. In summary, this is a discrete, deterministic, and high dimensional domain with well known structure in the solution.

In our example we use the eight disc domain and $\gamma = 0.99$. We also assume that TTree has been supplied with solutions to the three seven disc problems. These are policies that move the seven smallest discs, referred to as the 7 disc stack, onto a particular peg. These macros choose uniformly among the base level actions when the 7 disc stack is already on the appropriate peg. The complete set of abstract actions is shown in Table 5.3.

5.4.1 Building the abstract SMDP

Initially the algorithm has the entire state space as a single abstract state. The first thing the algorithm does is sample some trajectories.

Sampling Trajectories

TTree starts by choosing random start points for the trajectories. For our example, we assume the algorithm chooses the points in Table 2.1. We reprint that table here as Table 5.4.

Action	Effect
Generated abstract actions	
$\bar{a}_0$	Perform action a_0 in all states
$\bar{a}_1$	Perform action a_1 in all states
$\bar{a}_2$	Perform action a_2 in all states
$\bar{a}_3$	Perform action a_3 in all states
$\bar{a}_4$	Perform action a_4 in all states
$\bar{a}_5$	Perform action a_5 in all states
$\bar{a}_r$	Choose uniformly from $\{a_0, \ldots, a_5\}$ in all states
Supplied abstract actions	
$\bar{a}_{7P_0}$	If the 7 disc stack is on P_0 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_0.
$\bar{a}_{7P_1}$	If the 7 disc stack is on P_1 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_1.
$\bar{a}_{7P_2}$	If the 7 disc stack is on P_2 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_2.

Table 5.3: The abstract set of actions in the Towers of Hanoi domain

Trajectories are then taken from each start point with each action. Consider the first sample point, s_1. From this location, action a_1 solves the problem in a single step and receives reward. In our set of abstract actions, $\bar{a}_1$ does the same thing. In addition, $\bar{a}_{7P_2}$ also performs that base level action and solves the problem. $\bar{a}_r$ has one chance in six of performing the right action in this state. Even if it does not solve the problem with its first move, it performs a random walk near the solution and so has a high probability of solving the problem.

None of the other abstract actions solves the problem from s_1. In detail; $\bar{a}_{7P_0}$ and $\bar{a}_{7P_1}$ both move the agent away from the goal. Once they have moved the 7 disc stack to the appropriate peg, they perform a random action, and then fix the stack again if necessary: they never reach the solution. There are three base level actions that are illegal. The corresponding abstract actions simply stop with deterministic self-transitions after one step. There are two other legal base actions apart from the one that solves the problem. These move the

Example state ID	Disks on peg		
	P_0	P_1	P_2
s_1	D_1		$D_2D_3D_4D_5D_6D_7D_8$
s_2	$D_1D_4D_7$	D_2D_5	$D_3D_6D_8$
s_3	$D_1D_4D_7$	$D_2D_5D_8$	D_3D_6
s_4	$D_1D_4D_8$	$D_2D_5D_7$	D_3D_6

Table 5.4: A set of sample states in the Towers of Hanoi domain.

top disc on pegs 0 and 2 respectively to peg 1. These actions do not solve the problem. Furthermore, in the Towers of Hanoi domain performing any action leaves the agent in a state where performing the same action again is illegal, hence the generated abstract actions stop after two steps with a deterministic self-transition.

Having considered the resulting state of these trajectories, let us consider how long they take to run. $\bar{a}_1$ and $\bar{a}_{7P_2}$ both solve the problem in one step. The solution state is absorbing, so both trajectories stop immediately. The generated deterministic abstract actions that perform 'illegal' actions and so perform deterministic self-transitions likewise generate single-transition trajectories. The two generated deterministic abstract actions that do not perform 'illegal' actions as their first action perform illegal actions for their second action; their trajectories are two steps long. The 'random' abstract action take a variable amount of time that depends upon the distance to the goal. $\bar{a}_{7P_0}$ and $\bar{a}_{7P_1}$ each run for MAXTIME time steps.

Now we consider the second starting point. This starting point is similar to s_1 except that the problem cannot be solved in a single step from here. $\bar{a}_1$ behaves like the other generated abstract actions and only $\bar{a}_{7P_2}$ solves the problem efficiently. $\bar{a}_r$ is the other abstract action that might solve the problem. If it does so, the trajectory is likely to be fairly long.

The third and fourth starting points behave in similar ways. Like the second starting point, there is no base level action that solves the problem. Again, the random action might solve the problem, but it is unlikely, and we expect that if a solution is found it is significantly longer than the solution found from start point 2. This is simply due to the fact that the length of the shortest solution from point 2 is shorter than the length of the shortest solution from points 3 and 4.

None of the supplied macros solves the problem from points 3 or 4. If $\bar{a}_{7P_2}$ is selected, then D_8 is never moved onto P_2. If one of the other macros is selected, then D_8 might be moved onto P_2 at some point but the 7 disc stack is never moved to P_2.

We can generalize these results to classes of start points. Any start point with D_8 on P_2 reaches the goal state using $\bar{a}_{7P_2}$. Any start point with the D_8 disc on P_0 or P_1 is not solved by any macro, with the possible exception of the random macro. In a normal execution, TTree is set up to sample enough points that these regions are clearly outlined in the data.

The abstract transition function

Having sampled these trajectories, TTree uses them to construct an abstract SMDP. All the trajectories either reached the absorbing goal state or ended in the same abstract state they started in (there is only one state). The abstract transition function is as follows: The generated abstract action that solves the problem from start state one solves the abstract SMDP with low probability, $p = 0.25$ with just the four example start points, and self transitions the rest of the time. The random abstract action solves the problem with decreasing frequency and increasing trajectory length as the start state moves away from the goal state. $\bar{a}_{7P_2}$ solves the problem from s_1 and s_2. More generally, it solves the problem from any start state where D_8 is on P_2. It self-transitions the rest of the time. The other abstract actions all self-transition. Figure 5.1 shows the abstract state transition diagram. We have labelled the transitions with γ^t rather than t, as this is the value that is averaged when forming the transition function. Note that the random transition is marked with the maximum possible discount factor and reward. We would expect that they would both be significantly lower for any particular sample. It takes $\bar{a}_{7P_2}$ 116 steps to solve the problem from state s_2, but only 1 step from state s_1. These times give $\gamma^t \approx 0.31$, $r \approx 15.7$ and $\gamma^t = 0.99$, $r \approx 50$. The transition shown from the abstract state to the absorbing state uses weighted average values, $\gamma^t = 0.65$ and $r \approx 32.87$.

This abstract SMDP can then be solved (with the additional constraint that the absorbing state is assumed to have value 0). The resulting $\bar{Q}$ values are 0 for all abstract actions excepting $\bar{a}_{7P_2}$ and $\bar{a}_r$. $\bar{a}_{7P_2}$ has a $\bar{Q}$ value of 32.87. $\bar{a}_r$ has a $\bar{Q}$ value smaller than that, the exact value depending upon the random trajectory length. These values cause the algorithm to select $\bar{a}_{7P_2}$ as the action to perform in all non-absorbing states. This policy is optimal in approximately $1/3$ of the states.

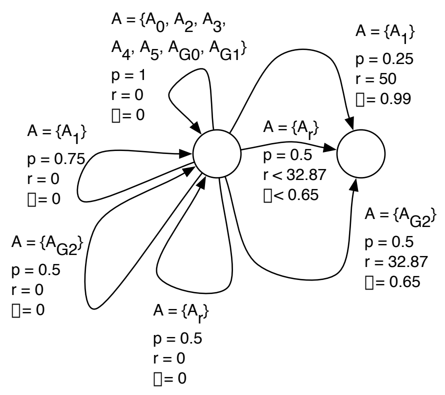


Figure 5.1: Example abstract state transition diagram for the Towers of Hanoi domain

5.4.2 Refining the abstract state space

Having formed a policy, the algorithm now attempts to refine the state. Each of the trajectories has a value assigned to the start state, $\hat{T}_s \leftarrow r + \gamma^t \bar{V}(\bar{s'})$ where r and t are the recorded values for the transition, and $\bar{V}(\bar{s'})$ is the value associated with the resulting abstract state for the trajectory.

In the current example, the value of the trajectory happens to equal the reward for the trajectory because all the transitions either have $\gamma^t = 0$ or transition into the absorbing state.

If we added more random start states then we would see a pattern similar to the classes of trajectories mentioned in Section 5.4.1 above. The $\hat{T}$ values of the trajectories for $\bar{a}_r$ are high near the goal state, and decrease quite quickly as you move away from the goal state. Any other trajectory starting with D_8 on P_0 or P_1 has $\hat{T} = 0$. Any trajectory for $\bar{a}_{7P_2}$ starting with D_8 on P_2 reaches the goal. These trajectories have $\hat{T}$ values that increase as the trajectory length decreases. These values are higher than the values for $\bar{a}_r$. Other trajectories have $\hat{T} = 0$ except the the one that starts in state s_1 and uses action $\bar{a}_1$ which has $\hat{T} = 50$.

For a reasonable statistical test, you need more than four start states. With a large enough set, the algorithm finds a distinction between states in which D_8 is on P_2 and other states. The resulting abstract state tree is shown in Figure 5.2.

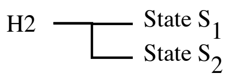


Figure 5.2: Example abstract state tree for the Towers of Hanoi domain

5.4.3 Further details

Having divided the state space, we now have two abstract states, $\bar{s}_1$, and $\bar{s}_2$. Moreover, we do not know when or even if the trajectories previously sampled cross between the states. We discard those trajectories and sample some new ones.

The generated abstract actions do not change their behavior much, so we concentrate on the supplied abstract actions.

In abstract state $\bar{s}_2$, D_8 is on P_2. From any base state in this abstract state, $\bar{a}_{7P_2}$ reaches the goal. Neither $\bar{a}_{7P_0}$ nor $\bar{a}_{7P_1}$ reaches the goal in this abstract state.

In abstract state $\bar{s}_1$, D_8 is either on P_0 or P_1. If D_8 is on P_0 then $\bar{a}_{7P_0}$ moves other discs on top of D_8. Once the other discs are stacked on P_0, $\bar{a}_{7P_0}$ chooses a random action. This action can only move D_1 or choose an illegal action.

However, if action $\bar{a}_{7P_1}$ is chosen then all the discs except D_8 are moved to P_1. Once in this state, $\bar{a}_{7P_1}$ chooses a random base-level action. This has a one in six chance of moving D_8 to P_2. That moves the problem into abstract state $\bar{s}_2$, from whence it can be solved. The other two legal base level actions move D_1 to another peg. $\bar{a}_{7P_1}$ simply moves it straight back and makes more random moves until $\bar{s}_2$ is reached. If the D_8 is on P_1 then symmetric situation arrises – $\bar{a}_{7P_0}$ moves to state $\bar{s}_2$.

Sampling the transition function and solving the abstract SMDP leads to the following policy: In $\bar{s}_2$, $\bar{a}_{7P_2}$ is chosen and solves the problem, and in $\bar{s}_1$, $\bar{a}_{7P_0}$ and $\bar{a}_{7P_1}$ are both equally effective – they each lead to state $\bar{s}_2$ half the time and self transition the other half of the time.

The $\hat{T}$ values in state $\bar{s}_1$ clearly indicate the regions where $\bar{a}_{7P_0}$ and $\bar{a}_{7P_1}$ are effective. Dividing $\bar{s}_1$ based on the location of D_8 separates those $\hat{T}$ values; this is the split that TTree introduces next.

At this point, the algorithm has three abstract states, one for each position of D_8. Once the transition function is sampled and the SMDP solved, there is one clearly superior abstract action in each state. Looking ahead a little to the empirical results, this state de-

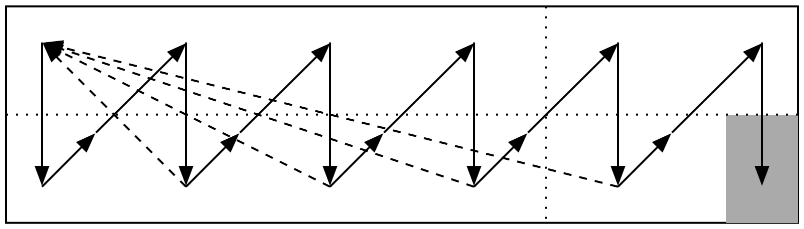


Figure 5.3: An SMDP with a single abstract state and two prospective divisions of that abstract state

composition corresponds to the peak in expected reward at 150 000 samples in Figure 5.4a. There are only two base level states where the action in non-optimal. These are the two states when D_8 is *not* on P_2 and the other discs are stacked such that the supplied abstract action chooses randomly amongst the base level actions.

In addition to these two states not having optimal actions, the actions in the other states have not been *proven* optimal by the algorithm; the $\hat{T}$ values across the states are not constant. In order to find a provably optimal policy, TTree must continue dividing the state. This division of the state does not necessarily monotonically improve the policy found by TTree.

5.5 An Example of a Problematic State Division

Consider the example shown in Figure 5.3. This figure shows a two dimensional state space. The grey region on the right hand side of the state space is absorbing with a reward of 50 for any action that enters that space. No other transitions in the space carry any reward. There are two actions. In the top half of the state space, both actions move the agent down in the same way. In the bottom half of the state space things are more complex. Action $\bar{a}_0$, shown with solid arrows, moves the agent up and to the right. It does this in a way that takes two steps to reach the top half of the state space. In the bottom half of the state space action $\bar{a}_1$, shown with dashed arrows, moves the agent to the top left of the state space.

Also shown as dotted lines are two potential divisions of this state space. One divides the right of the state space from the left of the state space, and the other divides the top from the bottom of the state space. With the state space undivided, action $\bar{a}_0$ has trajectories that reach the absorbing region on the right, whereas action $\bar{a}_1$ loops without gaining any reward on the left hand edge of the state space. Action $\bar{a}_0$ is chosen as the optimal action in the abstract SMDP.

If we introduce the first of the two possible divisions and split the right hand side of the state from the left, then the optimal actions in the abstract SMDP do not change. Action $\bar{a}_0$ still moves the agent to the right and to the reward, and action $\bar{a}_1$ moves the agent to the left. In fact as long as the state is divided based on the x value of the state then it does not matter at which x value the split occurs – action $\bar{a}_0$ is still the optimal action in each of the new abstract states.

If we introduce the second of the two possible divisions, and divide the top from the bottom of the state space, then we see that the optimal policy in the abstract SMDP is worse after the split than before. To see this, let us construct the transition function for the abstract SMDP. There are two abstract states, the top state, $\bar{s}_t$, and the bottom state, $\bar{s}_b$. In $\bar{s}_t$, both actions have the same transition function that moves the agent either into $\bar{s}_b$ or to the reward in a single step. In $\bar{s}_b$ the actions have different effects. $\bar{a}_0$ moves the agent into $\bar{s}_t$ in two steps, whereas $\bar{a}_1$ takes only one step to reach $\bar{s}_t$. Neither action gets any direct reward. Because $\bar{s}_t$ has positive value, and action $\bar{a}_1$ gets the agent to $\bar{s}_t$ faster than action $\bar{a}_0$, action $\bar{a}_1$ is selected in $\bar{s}_b$. This is far from optimal in the non-abstract problem.

What is happening is that information is lost on abstract state transitions. Often this is good because the information that is discarded is irrelevant. If the state is divided in the wrong way then important information can be lost. This is what happens in the example above. When the state is divided into top and bottom regions, the x location of the agent is lost each time there is a state transition. This makes the action that flips between abstract states faster look better than the one that flips between abstract states more slowly, but which moves the agent to the right.

TTree usually makes the correct decision when dividing states. The $\hat{T}$ values estimate how much discounted reward an agent receives for performing an action from a particular point. In the example above, the $\hat{T}$ values for action $\bar{a}_0$ would decrease from right to left across the state. This is the largest variation and a split dividing the left of the state from the right of the state would be introduced first.

TTree can make mistakes when there are not enough $\hat{T}$ values sampled in a region, particularly if the region has high dimensionality. In this case, random variation can cause less desirable splits to be chosen first. It is important to note that even when this occurs, future splits are introduced that allow the optimal policy to be found.

5.6 Empirical Results

We evaluated TTree in a number of domains. For each domain the experimental setup was similar. We compared mainly against the Prioritized Sweeping algorithm of Moore and Atkeson (1993). The reason for this is that, in the domains tested, Continuous U Tree was ineffective as the domains do not have much scope for normal state abstraction. It is important to note that Prioritized Sweeping is a *certainty equivalence* algorithm. This means that it builds an internal model of the state space from its experience in the world, and then solves that model to find its policy. The model is built without any state or temporal abstraction and so tends to be large, but, aside from the lack of abstraction, it makes very efficient use of the transitions sampled from the environment.

The experimental procedure was as follows. There were 15 learning trials. During each trial, each algorithm was tested in a series of epochs. At the start of their trials, Prioritized Sweeping had its value function initialized optimistically at 500, and TTree was reset to a single leaf. At each time step Prioritized Sweeping performed 5 value function updates. At the start of each epoch the world was set to a random state. The algorithm being tested was then given control of the agent. The epoch ended after 1000 steps were taken, or if an absorbing state was reached. At that point the algorithm was informed that the epoch was over. TTree then used its generative model to sample trajectories, introduce one split, sample more trajectories to build the abstract transition function, and update its abstract value function and find a policy. Prioritized Sweeping used its certainty equivalence model to update its value function and find a policy. Having updated its policy, the algorithm being tested was then started at 20 randomly selected start points and the discounted reward summed for 1000 steps from each of those start points. This was used to estimate the expected discounted reward for each agent's current policy. These trajectories were not used for learning by either algorithm. An entry was then recorded in the log with the number of milliseconds spent by the agent so far this trial (not including the 20 test trajectories), the total number of samples taken by the agent so far this trial (both in the world and from the generative model), the size of the agent's model, and the expected discounted reward measured at the end of the epoch. For Prioritized Sweeping, the size of the model was the number of visited state/action pairs divided by the number of actions. For TTree the size of the model was the number of leaves in the tree. The trial lasted until each agent had sampled a fixed number of transitions (which varied by domain).

The data was graphed as follows. We have two plots in each domain. The first has the number of transitions sampled from the world on the x-axis and the expected reward on the

y-axis. The second has time taken by the algorithm on the x-axis and expected reward on the y-axis. Some domains have a third graph showing the number of transitions sampled on the x-axis and the size of the model on the y-axis.

For each of the 15 trials there was a log file with an entry recorded at the end of each epoch. However, the number of samples taken in an epoch varies, making it impossible to simply average the 15 trials. Our solution was to connect each consecutive sample point within each trial to form a piecewise-linear curve for that trial. We then selected an evenly spaced set of sample points, and took the mean and standard deviation of the 15 piecewise-linear curves at each sample point. We stopped sampling when any of the log files was finished (when sampling with time on the x-axis, the log files are different lengths).

5.6.1 Towers of Hanoi

The Towers of Hanoi domain is well known in the classical planning literature for the hierarchical structure of the solution; temporal abstraction should work well. Described in Section 2.6.3 on page 31, it consists of 3 pegs, on which sit N disks. Each disk is of a different size and they stack such that smaller disks always sit above larger disks. There are six actions which move the top disk on one peg to the top of one of the other pegs. An illegal action, trying to move a larger peg on top of a smaller peg, results in no change in the world. The object is to move all the disks to a specified peg; a reward of 100 is received in this state. All base level actions take one time step, with $\gamma = 0.99$. The decomposed representation we used has a boolean variable for each disk/peg pair. These variables are true if the disk is on the peg.

Figure 5.4 shows a comparison of Prioritized Sweeping and TTree. In Figure 5.4b the TTree data finishes significantly earlier than the Prioritized Sweeping data; TTree takes significantly less time per sample. Continuous U Tree results are not shown as that algorithm was unable to solve the problem. The problem has 24 state dimensions and Continuous U Tree was unable to find an initial split.

The Towers of Hanoi domain had size $N = 8$. We had a discount factor, $\gamma = 0.99$. TTree was given policies for the three $N = 7$ problems, the complete set of abstract actions is the same as that shown in Table 5.3 on page 96. The TTree constants were, $N_a = 20, N_l = 20, N_t = 1$ and MAXSTEPS $= 400$. Prioritized Sweeping used Boltzmann exploration with carefully tuned parameters (γ was also tuned to help Prioritized Sweeping). The tuning of the parameters for Prioritized Sweeping took significantly longer than for TTree.

We also tested Continuous U Tree and TTree on smaller Towers of Hanoi problems without additional macros. TTree with only the generated abstract actions was able to solve more problems than Continuous U Tree. We attribute this to the fact that the Towers of Hanoi domain is particularly bad for U Tree style state abstraction. In U Tree the same action is always chosen in a leaf. However, it is never legal to perform the same action twice in a row in Towers of Hanoi. TTree is able to solve these problems because the, automatically generated, random abstract action allows it to gather more useful data than Continuous U Tree.

In addition, the transition function of the abstract SMDP formed by TTree is closer to what the agent actually sees in the real world than the transition function of abstract SMDP formed by Continuous U Tree. TTree samples the transition function assuming it might take a number of steps to leave the abstract state. Continuous U Tree assumes that it leaves the abstract state in one step. This makes TTree a better anytime algorithm.

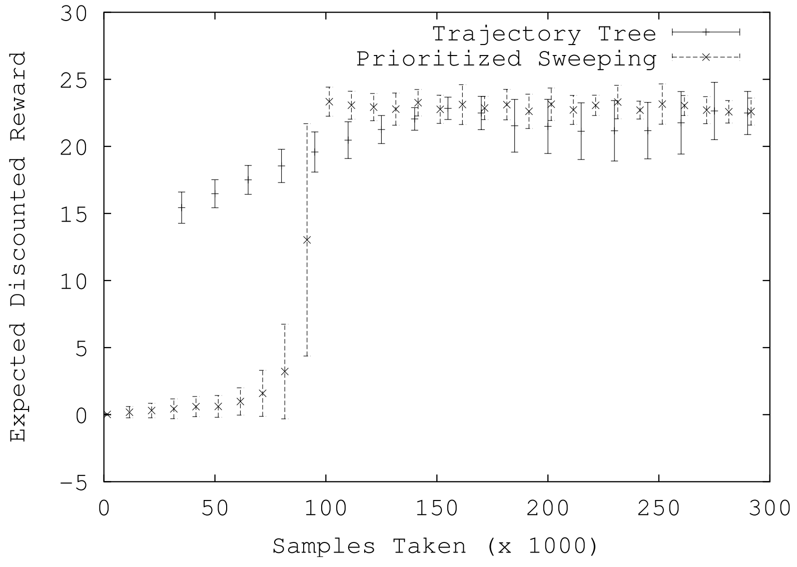
5.6.2 The taxi domain

The taxi domain is described in Section 2.6.5 on page 35. We supplied TTree with four abstract actions that move the agent to the four taxi stands and then performs the pick up and put down actions randomly. The various constants were $\gamma = 0.99, N_a = 80, N_l = 80, N_t = 1$ and MAXSTEPS $= 500$. The results for the taxi domain are shown in Figure 5.5.

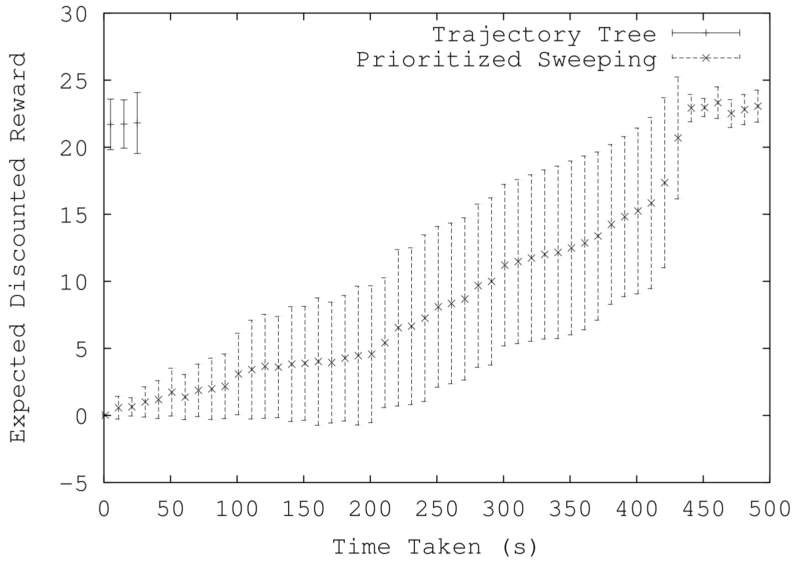
5.6.3 The rooms domains

In the introduction, and then more formally in Section 2.6.4 on page 32, we described walking robot domain. Figures 2.6 and 2.7, reprinted here as Figures 5.6 and 5.7, show some rooms for the robot to walk through. In each case there is a reward of 100 in the lower right of the world.

When solving the smaller of the two worlds, shown in Figure 5.6, TTree was given abstract actions that walk in the four cardinal directions: north, south, east and west. These are the same actions described in the introduction, *e.g.* Tables 1.1 and 1.2. The various constants were $\gamma = 0.99, N_a = 40, N_l = 40, N_t = 2$ and MAXSTEPS $= 150$. Additionally, the random abstract action was not useful in this domain, so I removed it. The other generated abstract actions, one for each base level action, remained. The results for the

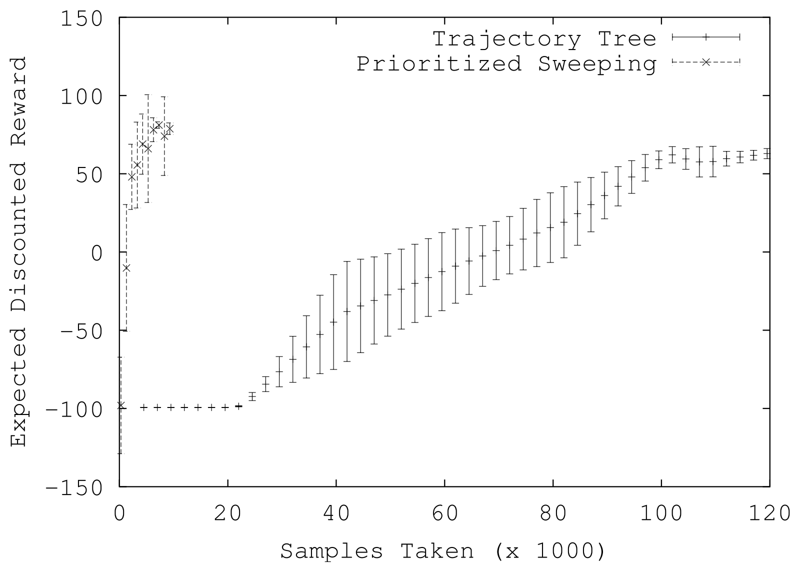


(a)

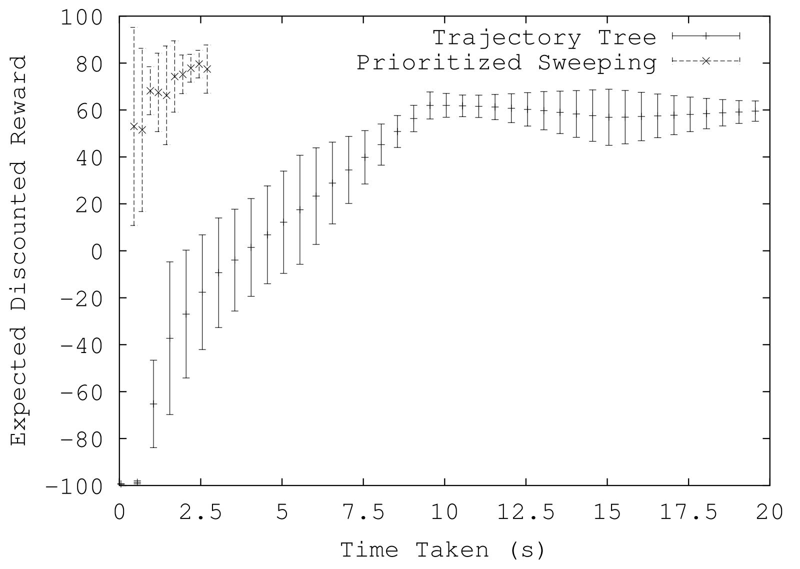


(a)

Figure 5.4: Results from the Towers of Hanoi domain. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples.



(a)



(b)

Figure 5.5: Results from the Taxi domain. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples.

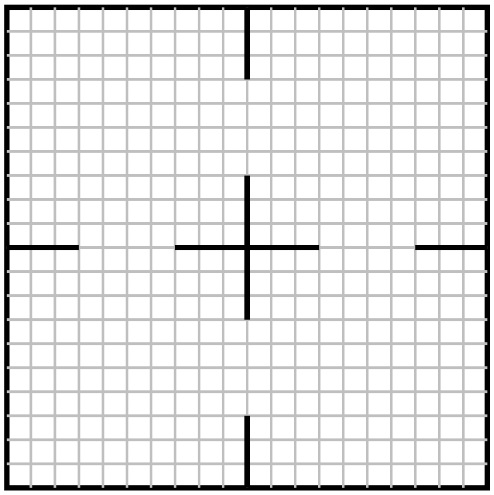


Figure 5.6: A set of four 10×10 rooms for our robot to walk through.

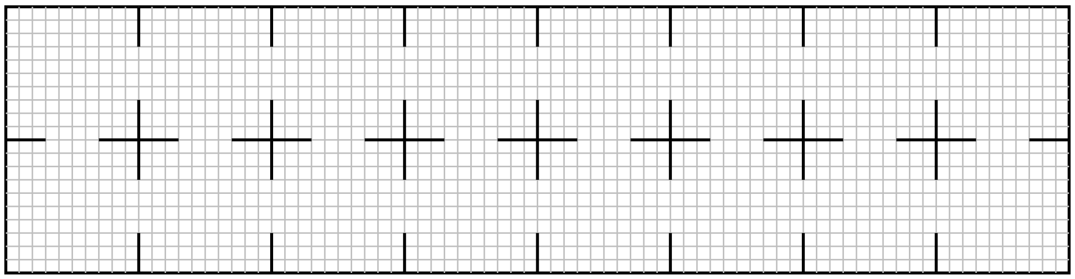


Figure 5.7: A set of sixteen 10×10 rooms for our robot to walk through.

small rooms domain are shown in Figure 5.8.

When solving the larger world, shown in Figure 5.7, we gave the agent three additional abstract actions above what was used when solving the smaller world. The first of these was a ‘stagger’ abstract action, shown in Table 5.5. This abstract action is related to both the random abstract action and the walking actions: it takes full steps, but each step is in a random direction. This improves the exploration of the domain. The other two abstract actions move the agent through all the rooms. One moves the agent clockwise through the world and the other counter-clockwise. The policy for the clockwise abstract action is shown in Figure 5.9. The counter-clockwise abstract action is similar, but follows a path in the other direction around the central walls.

The results for this larger domain are shown in Figure 5.10. The various constants were $\gamma = 0.99, N_a = 40, N_l = 40, N_t = 1$ and MAXSTEPS $= 250$. Additionally I modified the coefficient on the policy code length in the MDL coding to be 10 instead of 20.

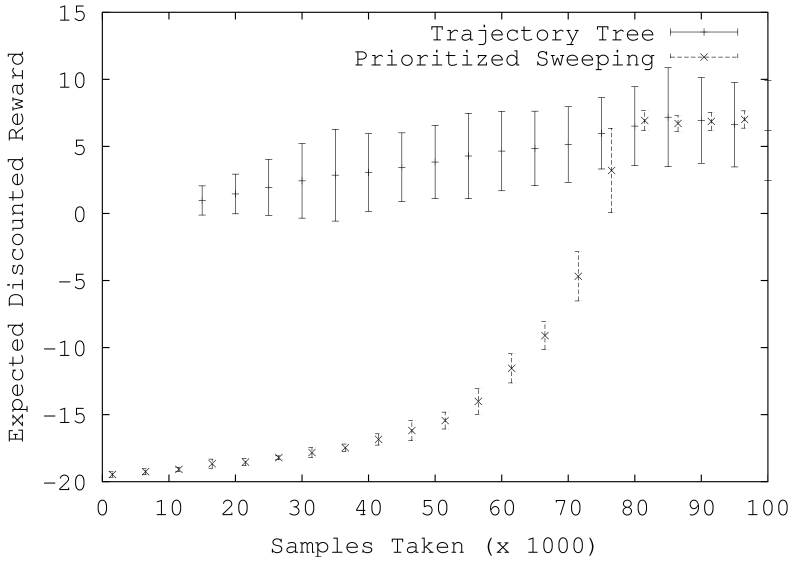
5.6.4 Discussion

There are a number of points to note about the TTree algorithm. Firstly, it generally takes TTree significantly more data than Prioritized Sweeping to converge, although TTree performs well long before convergence. This is unsurprising. Prioritized Sweeping is remembering all it sees, whereas TTree is throwing out all trajectories in a leaf when that leaf is split. For example, all data gathered before the first split is discarded after the first split.

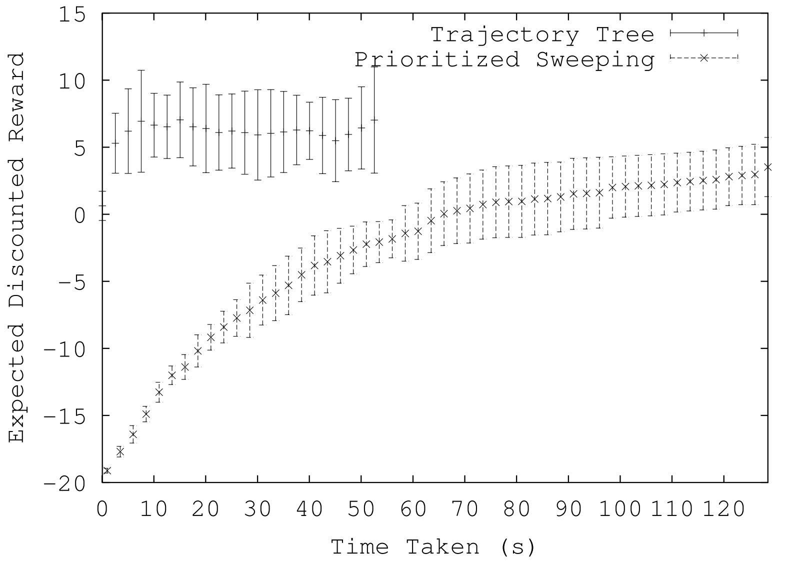
However, TTree is significantly faster that Prioritized Sweeping in real time in large domains (see Figures 5.4b, 5.8b and 5.10b). It performs significantly less processing on each data point as it is gathered and this speeds up the algorithm. It also generalizes across large regions of the state space. Figure 5.11 shows the sizes of the data structures stored by the two algorithms. Note that the y-axis is logarithmic. TTree does not do so well in small domains like the taxi domain.

Given this generalization, it is important to note why we didn’t compare to other state abstraction algorithms. The reason is because other state abstraction algorithms do not have a temporal abstraction component and so cannot generalize across those large regions. *e.g.* Continuous U Tree performs very poorly on these problems.

The next point we would like to make is that the abstract actions help TTree avoid negative rewards even when it hasn’t found the positive reward yet. In the walking robot domain, the agent is given a small negative reward for attempting to move its legs in an



(a)



(b)

Figure 5.8: Results from the walking robot domain with the four room world. (a) A plot of expected reward vs. number of transitions sampled. (b) Data from the same log plotted against time instead of the number of samples.

```
if $\Delta z < 0$ then {the right foot is in the air}
  if $\Delta x < 0$ then {left foot west of right foot}
    move the raised foot one unit west
  else if $\Delta x = 0$ then {right foot is same distance east/west as left foot}
    if $\Delta y < 0$ then {left foot south of right foot}
      move the raised foot one unit south
    else if $\Delta y = 0$ then {left foot is same distance north/south as right foot}
      lower the right foot
    else {left foot north of right foot}
      move the raised foot one unit north
    end if
  else {left foot east of right foot}
    move the raised foot one unit east
  end if
else if $\Delta z = 0$ then {both feet are on the ground}
  if $\Delta x = 0$ and $\Delta y = 0$ then {the feet are together}
    raise the left foot
  else
    raise the right foot
  end if
else {the left foot is in the air}
  if $\Delta x = 0$ and $\Delta y = 0$ then {the left foot is directly above the right foot}
    Move the raised foot north, south, east or west with equal probability
  else
    lower the left foot
  end if
end if
```

Table 5.5: The ‘stagger’ policy for taking full steps in random directions.

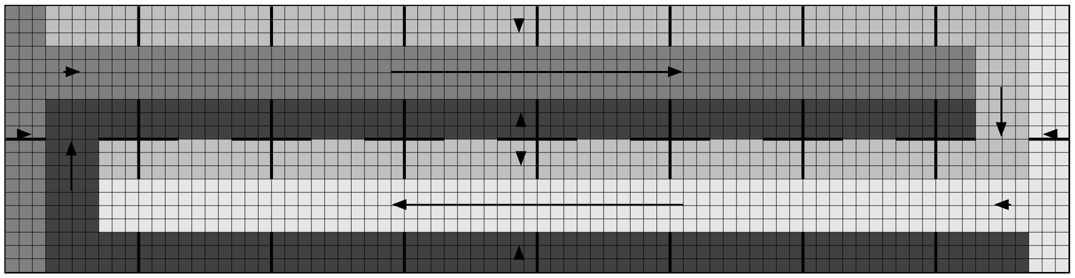


Figure 5.9: The clockwise tour abstract action.

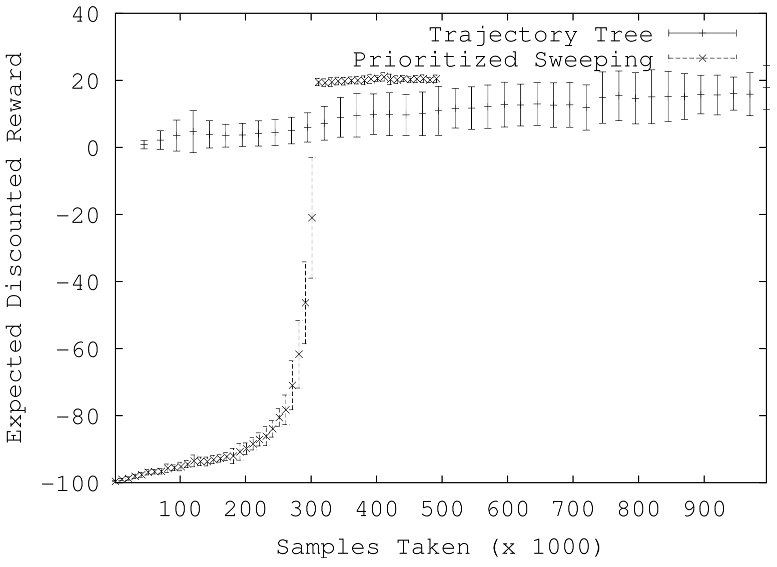
illegal manner. TTree notices that all the trajectories using the generated abstract actions receive these negative rewards, but that the supplied abstract actions do not. It chooses to use the supplied abstract actions and hence avoid these negative rewards. This is evident in Figure 5.8 where TTree's expected reward is never below zero.

The large walking domain shows a capability of TTree that we haven't emphasized yet. TTree was designed with abstract actions like the walking actions in mind where the algorithm has to choose the regions in which to use each abstract action, and it uses the whole abstract action. However TTree can also choose to use only part of an abstract action. In the large walking domain, we supplied two additional abstract actions which walk in a large loop through all the rooms. One of these abstract actions is shown in Figure 5.9. The other is similar, but loops through the rooms in the other direction.

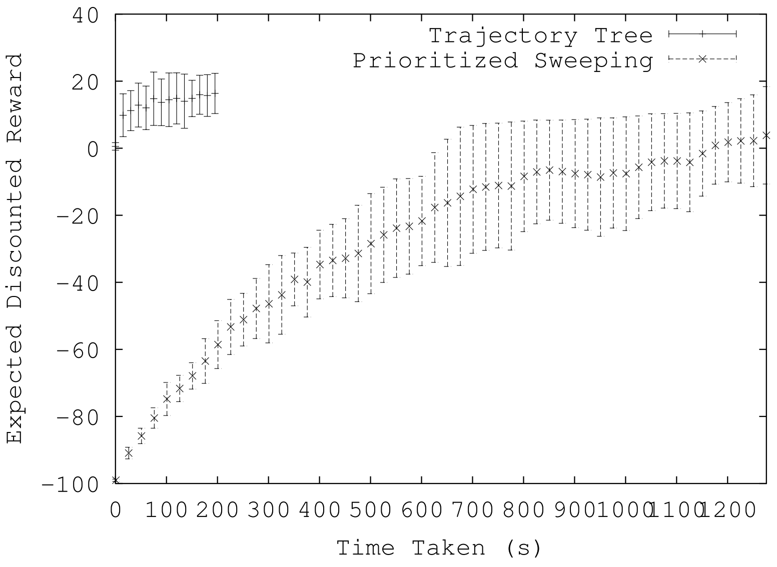
To see how TTree uses these 'loop' abstract actions, we show in Table 5.6 a small part of a tree seen while running experiments in the large walking domain. In the particular experiment that generated this tree there was a small, -0.1, penalty for walking into walls. This induces TTree to use the abstract actions to walk around walls, at the expense of more complexity breaking out of the loop to reach the goal. The policy represented by this tree is interesting as it shows that the algorithm is using part of each of the abstract actions rather than the whole of either abstract action. The abstract actions are only used in those regions where they are useful, even if that is only part of the abstract action.

This tree fragment also shows that TTree has introduced some non-optimal splits. If the values 78 and 68 were replaced by 79 and 70 respectively then the final tree would be smaller[3]. As TTree chooses its splits based on sampling, it sometimes makes less than optimal splits early in tree growth. The introduction of splits causes TTree to increase its

[3]The value 79 comes from the need to separate the last column to separate the reward. The value 70 lines up with the edge of the end rooms.

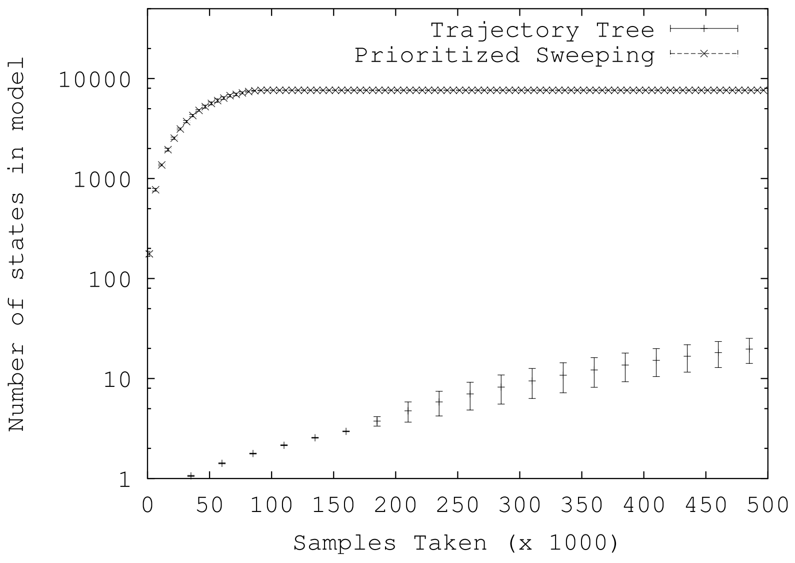


(a)

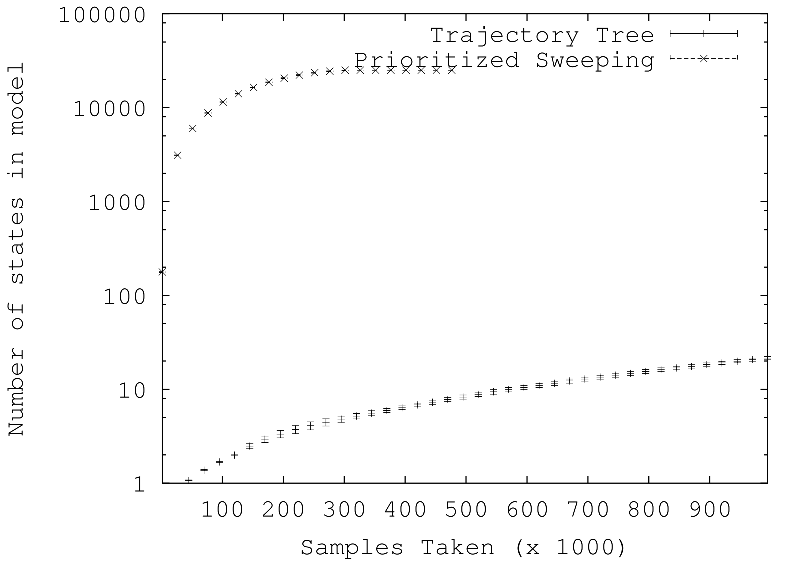


(b)

Figure 5.10: Results from the walking robot domain with the sixteen room world. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples.



(a)



(b)

Figure 5.11: Plots of the number of states seen by Prioritized Sweeping and the number of abstract states in the TTree model vs. number of samples gathered from the world. The domains tested were (a) the Towers of Hanoi domain, and (b) the walking robot domain with the sixteen room world. Note that the y-axis is logarithmic.

```
if $x < 78$ then
    if $x < 68$ then
        if $y < 10$ then
            perform the loop counter-clockwise abstract action
        else
            perform the loop clockwise abstract action
        end if
    else
        {Rest of tree removed for space}
    end if
else
    {Rest of tree removed for space}
end if
```

Table 5.6: Part of the TTree tree during the learning of a policy for the large rooms domain in Figure 5.7.

sample density in the region just divided. This allows TTree to introduce further splits to achieve the desired division of the state space.

The note above about adding a small penalty for running into walls in order to induce TTree to use the supplied abstract actions deserves further comment. The Taxi domain as described by Dietterich (2000) has a penalty of -10 for misusing the pick up and put down actions. It has a reward of 20 for successfully delivering the passenger. We found TTree had some difficulty with this setup. The macros we supplied chose randomly between the pick up and put down actions when the taxi was at the appropriate taxi stand. While this gives a net positive reward for the final move (with an expected reward of 10), it gives a negative expected reward when going to pick up the passenger. This makes the abstract action a bad choice on average. Raising the final reward makes the utility of the abstract actions positive and helps solve the problem.

When running our preliminary experiments in the larger walking domain, we noticed that sometimes TTree was unable to find the reward. This did not happen in the other domains we tested. In the other domains there were either abstract actions that moved the agent directly to the reward, or the random abstract action was discovering the reward. In the walking domain the random abstract action is largely ineffective. The walking motion is too complex for the random action to effectively explore the space. The abstract actions that walk in each of the four compass directions will only discover the reward if they are directly in line with that reward without an intervening wall. Unless the number of sample points made very large, this is unlikely. Our solution was to supply extra abstract actions whose goal was not to be used in the final policy, but rather to explore the space. In contrast to the description of McGovern (2002), where macros are used to move the agent through bottlenecks and hence move the agent to another tightly connected component of the state space, we use these exploration abstract actions to make sure we have fully explored the current connected component. We use these 'exploration' abstract actions to explore within a room rather than to move between rooms.

An example of this type of exploratory abstract action is the 'stagger' abstract action shown in Table 5.5. We also implemented another abstract action that walked the agent through a looping search pattern in each room. This search pattern covered every space in the room, and was replicated for each room. The stagger policy turned out to be enough to find the reward in the large walking domain and it was significantly less domain specific than the full search, so it was used to generate the results above.

5.7 Future Work

There are a number of dimensions along which the current TTree algorithm could be fruitfully extended. We list a number of these below.

As shown by Munos and Moore (1999), linear fits for the value function in the leaves of the tree can be useful. We had previously tested linear fits with Continuous U Tree and found them to be slow and to complicate the splitting criterion. We believe that it would be much easier to find a good splitting criterion for linear fits with trajectories.

As noted in Section 5.5, sometimes introducing a new split can degrade the performance of the policy in the short term, although future splits will bring the performance back to optimal. Linear fits in the leaves may help this. Another possible fix is to bias the splitting criterion away from splits that cut trajectories.

The choice of a Minimum Description Length (MDL) splitting criterion was made because we had previously been working with Lumberjack which used MDL and much code was reused. While the MDL splitting criterion does make it possible to combine the test for a change in $\hat{\pi}$ values and the test for a change in $\hat{V}$ values, we would like to investigate other splitting criteria. If linear splits are introduced this will almost certainly be required.

At the moment TTree searches for the best leaf to divide, divides it, samples more data, solves the SMDP and then searches for a new leaf to divide. Other algorithms, like Continuous U Tree, divide multiple leaves at once. It would be worth exploring what effect this has upon TTree.

All of the above changes are relatively small. One more ambitious change would be to try to find a way of randomly sampling a start state within a region, *without requiring teleportation*. This would allow TTree to work without a generative model.

5.8 Further Experiments

There have been two questions raised about the TTree algorithm. Firstly, the experiments above compare an algorithm with both state and temporal abstraction against an algorithm with neither; is it possible to separate the contributions of these abstraction techniques in TTree? Secondly, the experiments presented previously are mostly deterministic; how does a stochastic environment affect TTree?

5.8.1 Separating state and temporal abstraction

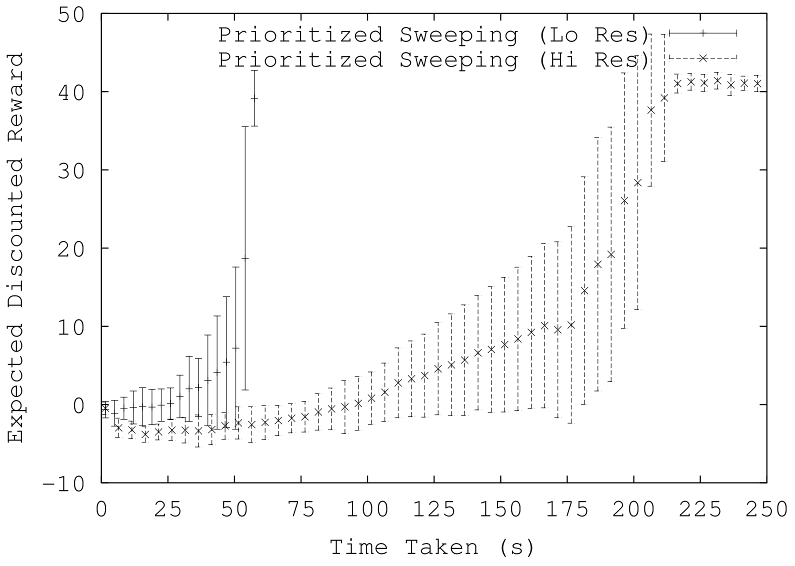
Separating state and temporal abstraction within TTree is non-trivial. In particular, the temporal abstraction within TTree requires the state abstraction to be effective, and hence the temporal abstraction cannot be tested without the state abstraction. It is, however, possible to test the state abstraction without the temporal abstraction. We modified the walking robot domain to remove element that requires temporal abstraction – the walking motion. The actions that moved the legs were replaced with four actions allowing movement in the four compass directions. This gives us a ‘sliding robot’ domain similar to the grid-worlds commonly used in reinforcement learning.

The domain had the same overall layout as the 4 room walking robot domain (see Figure 2.6). Additionally we wished to be able to vary the number of states in the domain without changing the value function. To achieve this we modified the domain to be a full SMDP. Each of the x, y locations in Figure 2.6 was defined to be 1 time step wide in the new domain, but was subdivided into either 16 (Lo Res, 4×4), or 36 (Hi Res, 6×6) states. The actions that move between states were set to take either $1/4$ (Lo Res) or $1/6$ (Hi Res) time units. TTree only used the automatically generated macros in this domain – it was only being used for state abstraction, not temporal abstraction. The x, y location of the robot was transformed into an attribute vector using the same techniques described for the walking robot domain, *i.e.* a list of boolean attributes $x < 1, x < 1.25, x < 1.5, \ldots$. TTree was then compared with Prioritized Sweeping in this new domain.

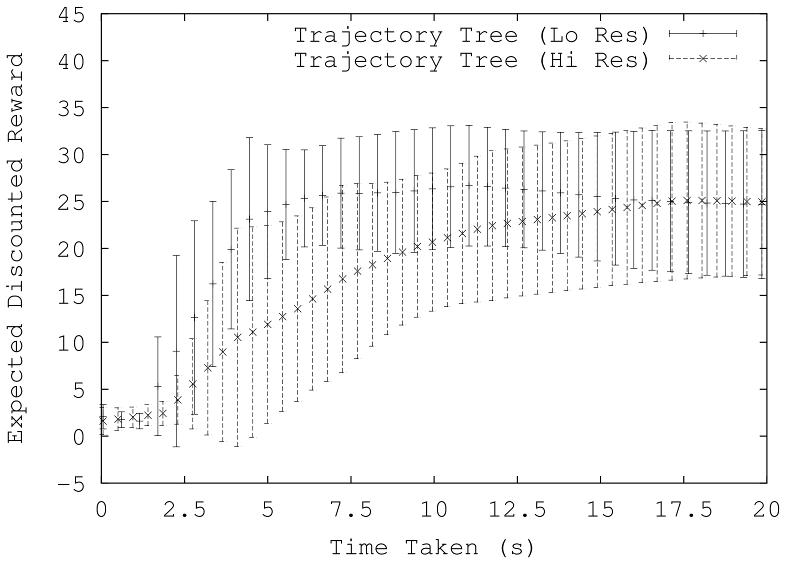
My predictions before running these experiments were that the increase in resolution would affect Prioritized Sweeping, but not TTree. The results of the experiments are shown in Figures 5.12 and 5.13. There are a number of points to notice in these graphs. In Figure 5.12a we see that Prioritized Sweeping is significantly affected by the increase in the number of states between the Lo Res and the Hi Res domains. In Figure 5.12b we see that TTree is not significantly affected, however there is a small decrease in efficiency as the problem size increases. The reason for this is obvious in hindsight: while both the $\hat{T}$ values and the learnt trees are similar in the Lo Res and Hi Res experiments, the trajectories themselves are 50% longer in the Hi Res experiments. These longer trajectories take more time to sample, but the increase is proportional to the change in length of each dimension separately, whereas the number of states is proportional to the product of the change in length of each dimension. Stated slightly differently, the trajectories increase in length proportionally to the increase in linear size of the domain, whereas the number of states increases proportionally to the increase in *area* of the domain – proportionally to the square of the increase in linear size.

Figure 5.13 shows the above results quite clearly, and also shows another somewhat surprising result: it looks like TTree is converging to an expected reward 30% lower than Prioritized Sweeping. There are two obvious questions: what is going on, and why is it more obvious here than in previous experiments? The answer is that TTree only divides the 'best' leaf in each iteration of the main loop, whereas other state abstraction algorithms divide every leaf that should be divided. TTree's method is not as effective. In light of these results a little more discussion is warranted.

Initially we made the choice to only introduce the best split in each iteration for two reasons: it allowed the implementation of TTree to build upon the tree library developed for Lumberjack, and it meant that other splits would be able to take advantage of the increased resolution and hence be more accurate. It should be noted that this choice is orthogonal to the main contribution of the TTree algorithm, temporal abstraction, and we mentioned investigating the choice in more detail in the future work section above. Unfortunately, these results show that introducing only one split for iteration seems to have a net negative effect on the algorithm. It introduces two problems. The first is simply that the algorithm takes longer to reach a given resolution. The increase in accuracy of the splits is not enough to



(a)



(b)

Figure 5.12: Results from the sliding robot domain. (a) A plot of expected reward vs. time taken for Prioritized Sweeping. (b) A plot of expected reward vs. time taken for TTree.

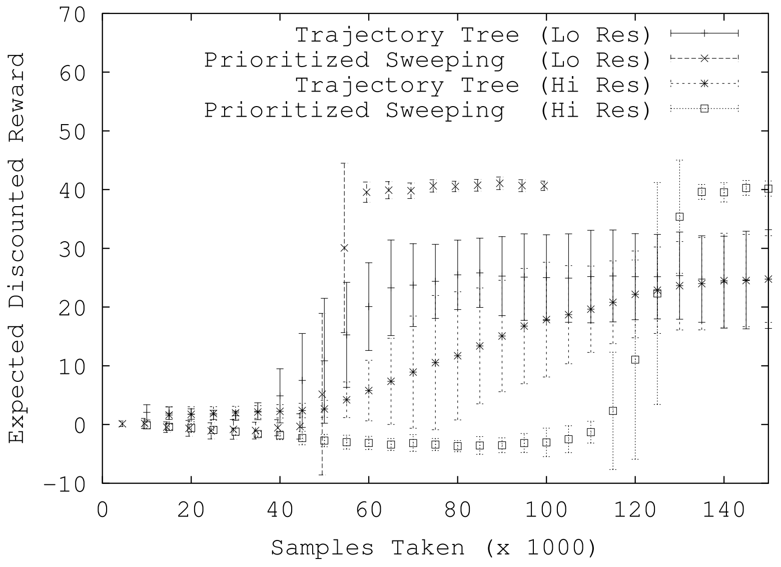


Figure 5.13: Results from the sliding robot domain. A plot of expected reward vs. number of sampled transitions.

offset the decrease in speed. The second is that TTree tends to increase resolution in one area at a time. If one area needs an increase in resolution more than others it is divided. After it is divided, more samples are taken in that region increasing the local sample resolution. This makes it likely that another split will be introduced in the same area. The end result is that TTree converges slowly in this case. While the results above suggest that TTree has converged, it has not. It is simply converging very slowly.

This slow convergence shows up more clearly in these experiments than others because of the representation of the state space. The Lo Res sliding domain and the walking domain are of similar size (the increased x, y resolution in the sliding domain replaces the state required for walking). However, the translation into attributes used to divide the state space is significantly different. In the walking domain, the increase in state-space size is represented as three extra attributes. In the sliding domain, the increase in state-space size is represented as an increase in resolution in the x, y plane. This is then translated into four-fold increase in the number of boolean attributes used to represent the plane. This massive difference in the number of attributes between two state spaces of approximately the same size explains the difference in results.

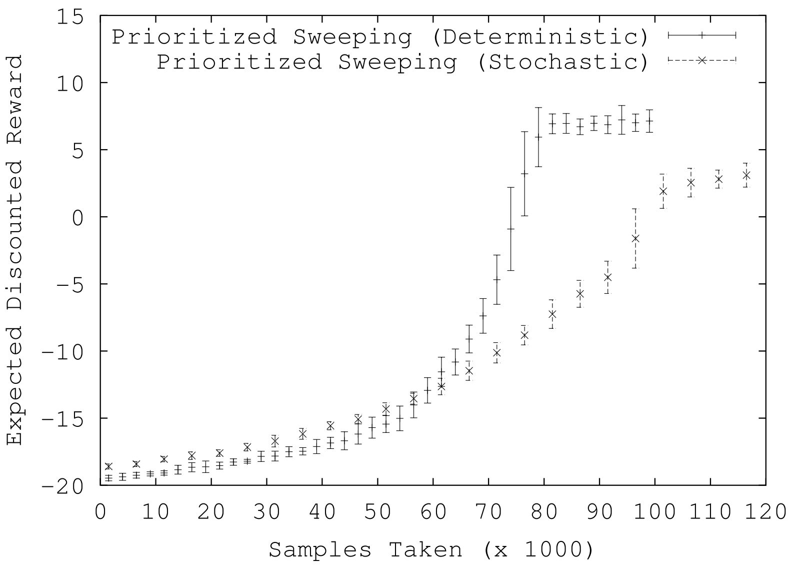


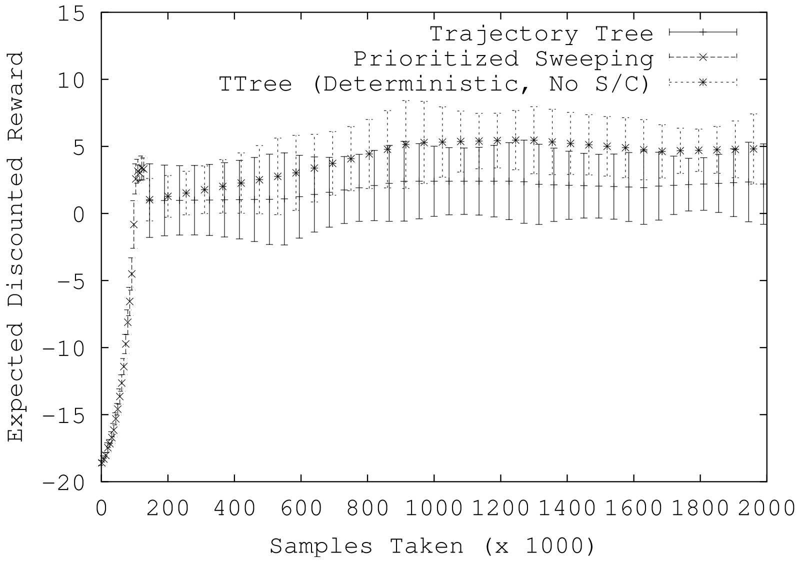
Figure 5.14: Results from the walking robot domain with the four room world. A plot of expected reward vs. number of sampled transitions.

5.8.2 Response to stochastic domains

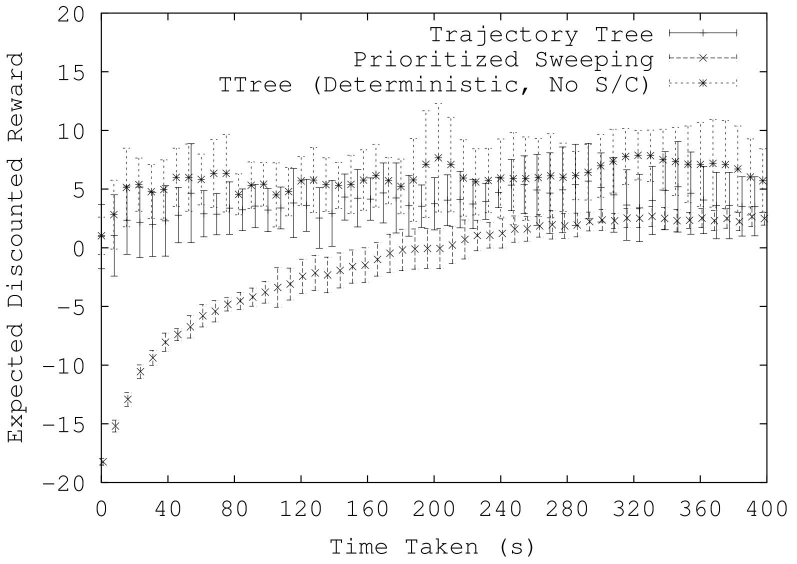
The second question raised about TTree was its response to stochastic domains. TTree allows a trajectory to terminate early when a deterministic self-transition is encountered. The domains we used for testing in previous experiments were designed to take advantage of this by making all self-transitions deterministic. Relaxing this requirement is going to significantly impact TTree. We predicted that it would not have much impact apart from the loss of the self-transition optimization.

To test this we modified the walking robot domain to move randomly with 10% probability. This change modifies the optimal expected discounted reward in the domain, as shown in Figure 5.14. The stochastic domain has an optimal expected discounted reward approximately 5 lower than the deterministic domain. We then made a modified TTree algorithm without the deterministic self-transition termination criterion for trajectories. Finally we compared the modified algorithm in the deterministic domain with the unmodified algorithm in the stochastic domain.

Figure 5.15 shows the results of this experiment. Note that both forms of the TTree algorithm are using significantly more samples than prior experiments – Prioritized Sweeping solves the problem before TTree has really gotten started. Also note that the two forms of TTree do differ in expected discounted reward, however it is explained by the difference in the expected discounted reward of the optimal policy between the stochastic and deterministic domains. It is also interesting to note that TTree is still faster than prioritized sweeping even if less efficient in terms of samples.



(a)



(b)

Figure 5.15: Results from the walking robot domain with the four room world. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples.

Chapter 6

Related Work

In earlier parts of this thesis we have referenced work that was closely related to ours, and work that we have built upon. In this chapter we review some of that research in more detail, and also look at other people's approaches to temporal abstraction in reinforcement learning.[1]

There are a number of dimensions along which we can classify this related work. For state abstraction we can ask if work uses a tree based representation? If so, is that representation a variable resolution tree, or a decision/regression tree? For temporal abstraction we can ask if the high level controller can interrupt a low level task (see Section 2.4.3, subroutines vs. polling)? Is the hierarchy strict: can a high level in the hierarchy skip the level immediately below it and access the lower levels of the hierarchy directly? Do the subtasks overlap in the state space or not? Is the hierarchy in the policy, the transition function or the value function? We classify each piece of work along the relevant dimensions.

We introduce this work out of chronological order. That is, we use some of the more well known and theoretically justified frameworks to explain some of the other work, even if the more theoretically justified frameworks came later.

6.1 State Abstraction

Gordon (1995) was the first person to introduce a class of function approximation techniques for reinforcement learning with known convergence guarantees. While individual

[1]We do not review related work in supervised learning in this chapter. We refer the reader back to Section 4.1.

convergent techniques were known prior to this, many techniques were also known to have difficulties in some cases. Gordon showed that for a function approximation technique to converge with a standard incremental update rule (irrespective of whether the final solution is actually useful) the function approximator must be a contraction mapping. That is, any change in the inputs to the function approximator cannot lead to a larger change in the outputs. An example of a function approximator that fails this test is linear extrapolation. There, small changes in training values at one point can lead to large changes in predicted value at other locations. Linear interpolation, however, is safe: the values between two interpolation points change less than the values of the points themselves. If a grid of interpolation points is imposed over the state space, then something related to state abstraction is achieved.

The G-Algorithm (Chapman and Kaelbling, 1991) was an early algorithm that used decision trees. It was used to control a computer game with an underlying continuous state space. The attributes of the computer game were manually discretized in an ego-centric manner. That is, the state was discretized in terms of distance from the agent, rather than in an absolute coordinate system. The G-Algorithm then built a tree, using the discretized attributes to discretize the state. Initially there was one large state in the world. The value of individual transitions was calculated in a manner similar to Continuous U Tree, and tests for variation were performed using a t-test. The attribute that had the most variation across its prior discretization was introduced as a split. Our Continuous U Tree algorithm, described in Chapter 3, is an extension of this algorithm to continuous spaces.

The U-Tree algorithm (McCallum, 1995) also extended the G-Algorithm. As well as attributes of the current state, U-Tree could introduce divisions in the tree based upon the attributes of prior states. This allowed U-Tree to handle partially observable MDPs as long as the memory length required in the controller was limited. U-Tree only handled discrete attributes.

The Parti-Game algorithm (Moore, 1994) is the first of the variable resolution algorithms we consider. It has a large number of similarities with our TTree algorithm. Parti-Game uses a pre-supplied controller that can "move greedily towards any desired state", although "there is no guarantee that a request to the greedy controller will succeed." Parti-game then partitions the state space using a tree and annotates each leaf with the direction the local controller should head to reach the next leaf. This is somewhat analogous to TTree in that where Parti-Game uses a low level controller, TTree uses supplied policies. Parti-Game also has a number of differences from TTree. Parti-game assumes a deterministic environment. It assumes a single goal region in the state space, rather than a set of rewards to maximize. This rules out policies that achieve maintenance goals.

The state abstraction algorithm of Munos and Moore (1999), which we will refer to as the influence-variance algorithm, is also quite closely related to TTree. However, again, there are a number of differences. Both algorithms use trees to discretize the state space. Both algorithms sample within the leaves and integrate the effects of actions from a sample point to the edge of the leaves. However, where TTree was designed more for discrete state spaces, the influence-variance algorithm was designed for continuous state spaces. Where TTree uses decision-tree style splitting, the influence-variance algorithm uses variable resolution splitting. Where TTree uses abstract actions, the influence-variance algorithm uses only base level actions.[2] Where TTree uses single valued 'flat' leaves, the influence-variance algorithm uses linear interpolation. Finally, the splitting criteria are quite different. The influence-variance algorithm uses variable resolution techniques that need to be able to decide "Do I increase resolution here?", whereas TTree tries to decide *how* to increase resolution as well.

The effect of the differences between the influence-variance algorithm and TTree deserves some more discussion. Many of the choices made above reflect the different focuses of the algorithms. The influence-variance algorithm provides much more accurate function approximation at a low level, while TTree was designed to support temporal abstraction. While both algorithms use trees, and integrate trajectories to the edge of the current leaf to build up transition functions over those trees, the different choices made reflect the different emphases. Firstly, single value leaves vs. linearly interpolated leaves: By using flat leaves TTree is able to supply its proof of correctness, whereas the influence-variance algorithm is heuristic and usually starts with some discretization to prime the splitting. Another difference is TTree's use of more accurate splitting. By using regression tree splits instead of variable resolution splits, TTree has higher effective resolution at intermediate stages of tree growth. This is useful as it helps the semantics of the abstraction match the base level SMDP (see Section 5.5).

There have been other techniques for function approximation apart from decision trees. The Residual Gradient Descent algorithm (Baird, 1995) was the first algorithm for using general gradient descent function approximation techniques with reinforcement learning (Baird used neural nets for his experiments). With the correct topology, these neural nets

[2]However, both algorithms use these actions in the same way and so this is more a difference in how the algorithms are used, rather than how they operate.

could have implemented a form of temporal abstraction. With a single hidden layer in the neural net only state abstraction is achieved.

Unrelated to the above is another class of state abstraction systems that do not try and perform their state abstraction online. Gordon (1995) noted that linear extrapolation is unstable when used for function approximation in traditional reinforcement learning value function updates. Boyan and Moore (1995, 1996) noted that if performed offline, these ’unstable’ approximation algorithms could be made stable. This culminated in the Batch Fit to Best Paths (BFBP) algorithm of Wang and Dietterich (2002). This the same overall approach used in our thesis. Lumberjack does not work well when naively combined with Continuous U Tree. The result is that we use it offline and then use the results of that offline decomposition to help with the next problem.

6.2 Temporal Abstraction

The *options* framework (Sutton et al., 1998; Precup, 2000) is a very clean definition of macro actions for reinforcement learning. In this framework, options are defined as policies with termination conditions. The termination conditions consist of a set of states, and for each of those states a probability of terminating. The options are used in an SMDP just as base level actions would be – they are called, they run until termination and then control is returned to the caller. Options are purely a solution to the temporal abstraction problem. They are designed to be orthogonal to state abstraction techniques. They are also not defined with hierarchy – they are used in addition to the base level actions rather than instead of base level actions and they often overlap. It is possible to learn the policy for an option given an SMDP. Usually the SMDP defined for an option has rewards at the terminating states, possibly in addition to the rewards in the original SMDP. One important point is that the policy learned for an option depends upon the rewards in the option’s SMDP. If there are large rewards at the termination states, then the agent accepts lower rewards on the way to the termination state in order to arrive there quickly. Like the termination states themselves, the SMDP used to learn an option is supplied by the user, although there are simple heuristics that give reasonable SMDPs for a set of termination states.

The work of McGovern and Barto (2001) and McGovern (2002) builds directly upon the options of Precup (2000). The original work on options defined the use of macros given pre-defined termination conditions. McGovern and Barto (2001) detects bottlenecks in the state space and uses these as the termination criteria of new options. McGovern makes the interesting point that options discovered this way increase the connectivity of the state space and can thereby make exploration more uniform. For example, in a gridworld with four rooms, the doorways are bottlenecks. It is more likely that the agent will find a random path between a pair of states within a single room than between a pair of states in two separate rooms. Adding options that move to the doorways increases the chance of moving between rooms.

The Hierarchical Distance to Goal (HDG) algorithm of (Kaelbling, 1993) defines "landmarks" which the agent heads towards (somewhat like the termination criterion of options). Specifically, the user supplies a set of landmarks and a distance metric. The Voronoi diagram is then used to partition the state space into a set of regions. Within each of these regions a policy is learned to reach the landmarks of each of the neighboring regions. A goal can be reached by planning over the abstract space defined by the landmarks. The agent heads either for the goal state directly, if it is close, or for the next landmark along its abstract path. This work is also interesting because the agent only reaches the intermediate landmarks by accident on the way to another landmark. Once the agent has moved into the region containing the landmark it is heading towards it moves one step in its abstract plan and aims for a new landmark: this is a polling system rather than a subroutine system.

The airports algorithm (Moore et al., 1999) is an extension of the HDG algorithm. It extends that algorithm by automatically generating the hierarchical decomposition of the state space (as well as having multiple levels of hierarchy and a slight re-formulation of the regions surrounding each airport/lighthouse). Like many of the hierarchical reinforcement learning algorithms, this algorithm is motivated by the requirement to solve a series of related tasks. In this formulation the tasks are related by having the same world dynamics. Each task has a different state as its goal.

Feudal reinforcement learning (Dayan and Hinton, 1993) also tiles the state space with regions. Like the airports algorithm there are multiple layers to the hierarchy. Unlike the airports algorithm, all the divisions at one level fit within the divisions at higher levels. At the lowest level the state space is tiled with managers that use base level actions to move in each direction. At the next level the space is tiled with a coarser grid. Each grid square is a manager that uses the managers at the level below to learn to move in each of the four directions. This hierarchy is built up until at the top level there is only one grid square. Feudal RL uses a neural net controller for each region to learn how to exit the region in each of the four cardinal directions. This was shown to slow down the solution of a single problem, but once the lower managers had learned policies, new problems could be solved relatively quickly.

The MaxQ algorithm (Dietterich, 1998, 2000) formalizes the strict hierarchy algorithms we have seen above. It takes a user-supplied decomposition of the SMDP and uses it to improve problem solving speed. The *task graph* is a strictly hierarchal decomposition of the action hierarchy for the current task. Each node in the task graph is either a primitive action, or a task that can be completed. Each task has as children the tasks or primitive actions it is allowed to use. As the task graph decomposes the action space of the problem, the *MaxQ graph* decomposes the value function of the problem. The result is a method for online learning using a pre-supplied decomposition of the action hierarchy. Dietterich (2000) notes that the final policy is required to follow the supplied action decomposition and is hence only hierarchically optimal, although once learned the value function can be executed in a polling manner giving some improvement towards true optimality. He also notes that state abstraction is very useful in combination with temporal abstraction, something we have found to be true in TTree.

The HexQ algorithm (Hengst, 2000, 2002) can be viewed as an extension of the MaxQ algorithm (Dietterich, 2000). Whereas MaxQ uses a pre-supplied hierarchical decomposition, HexQ generates a hierarchical decomposition and then solves it using MaxQ-like techniques. HexQ, like our work, requires a factored representation of the state space. HexQ then uses a heuristic to impose an ordering on the state variables. Variables that change more quickly are associated with the lower levels of the hierarchy. With this variable ordering in place, HexQ proceeds to build the hierarchy in a bottom up manner. In HexQ hierarchy is discovered in the state transition function, whereas in our work hierarchy is discovered in the policy. Unlike both Options and MaxQ, HexQ defines the termination criteria of abstract actions in terms of particular transitions, rather than particular states.

Singh's CQ-learning and H-DYNA algorithms (Singh, 1991, 1992b,a) can be viewed as early forms of the MaxQ algorithm. A hierarchy is supplied externally. The abstract actions in the hierarchy are constrained to have a single termination state. The problems being solved are constrained to be non-discounted, and hence finite. The value function for the combined MDP is simply the value functions of the parts each adjusted by a constant.

Dean and Lin (1995) describe a pair of algorithms for solving decomposed MDPs. These algorithms take a decomposition of a state space into regions and treat those regions as options. The algorithms the optimize the termination values of the options so that the MDP can be solved optimally in a strictly hierarchical manner.

The macros of Hauskrecht et al. (1998) are similar to the options framework, but with a number of changes. Specifically, Hauskrecht et al. (1998) integrate the use of state abstraction with their macro actions, and also impose a strict hierarchy on their abstract actions. The model relies upon the region based decomposition of Dean and Lin (1995). The state space is decomposed into regions, and the edges of the regions are used as the termination criteria for macros. There is some discussion of reuse of macros between problems. This is defined in terms of explicitly reused subsets of the state space without any generalization.

The Hierarchical Abstract Machines described by Parr and Russell (1998) and Parr (1998) are a way of constraining the solution to an MDP. Along with the MDP, a user supplies a series of state machines. These state machines can refer to each other, and hence form a hierarchy. The state machines partially specify a policy. They can be viewed as a set of constraints – only those policies consistent with the state machines are allowed. Parr then gives algorithms for turning the original MDP and constraints into a new, smaller MDP. This smaller MDP can be solved using traditional methods. The policies discovered for the smaller MDP are optimal among those policies that fit the supplied constraints.

The Skills algorithm (Thrun and Schwartz, 1995) attempts to learn a hierarchical decomposition of a state space. Like Lumberjack, the Skills algorithm uses a minimum description length methodology. Skills are defined while solving a set of problems in parallel, all of which have the same set of states and dynamics. A skill consists of a subset of those states and the action assignments for those states. The number of skills that the algorithm attempts to find is predefined. Description length is minimized when the solution policies of the problems have subsets of states with similar action assignments. That set of states and their action assignments are moved into a skill. The skills are used with a very simple top-level controller. There is a top level controller for each problem that has a probability of using each of the skills, or a full Q-table for the current problem. This top level controller is 'polled' at each time step, but the weight of a particular skill does not depend on the current state. The weights of all skills that are defined for the current state are normalized and a skill to execute is chosen stochastically using the normalized weights as probabilities.

The SPUDD algorithm (Hoey et al., 1999) uses function approximation to achieve a limited form of temporal abstraction. It uses Arithmetic Decision Diagrams (ADDs), related to the Ordered Binary Decision Diagrams of Bryant (1992), to represent the value function while it performs value iteration. Decision Diagrams are very similar to decision trees, but use a directed acyclic graph rather than a tree. They also require a specific ordering of the variables. With this ordering they are able to efficiently construct a compact canonical representation for functions. If there is a section of the value function, defined over a subset of the variables, that is exactly duplicated in two regions of the state space, then the ADD is usually able to merge the representation of the two regions, given an effective variable ordering. St-Aubin et al. (2000) extends this work to allow the regions to match approximately. This approximate matching allows the regions to differ by small amounts in many places, but it does not allow them to be offset from each other by a larger constant.

Nested Q Learning (Digney, 1996) is an algorithm which attempts to build a hierarchy of skills. Initially a set of 'skills' are defined, with each 'skill' being similar to an option with a termination criterion of making one state attribute have a particular value. This leads to a very large set of skills. These skills can use both the base level actions and each other, as actions. It is this recursive nature of the large number of pre-defined options that makes this hierarchical. In Digney (1998) the number of defined skills is limited to those whose endpoints have either high state count or high reinforcement.

Chapter 7

Conclusion

The goal of this thesis was to investigate the generation and use of hierarchy in reinforcement learning. We investigated the question: “How can the solution to one problem be processed to provide data for future problems?”

Our approach to this involved using decision/regression tree representations to divide the base state space into abstract states, and using policies for temporally abstract actions over those abstract states. Solving large problems involves solving smaller, related, problems first. The solutions to these problems are analyzed to extract repeated structure in the solutions. This is used to help solve larger problems.

We introduce one small modification of a previous algorithm, and two significantly new algorithms in this thesis. Continuous U Tree is only a small change from G-Algorithm or U Tree. It can generate its own abstract discrete state space from a continuous underlying state space. Its main purpose in this thesis is to introduce regression tree reinforcement learning. Lumberjack is a supervised learning algorithm that decomposes a function as it learns the representation for it. TTree is a significant extension of Continuous U Tree. As shown in Chapter 5, it is more effective than Continuous U Tree and additionally, it can make use of supplied abstract actions.

Lumberjack deserves further review. It is a supervised learning algorithm; given a series of examples of input-output pairs it attempts to model the generating function. Lumberjack does this using a tree based structure. By detecting, extracting and re-using duplicated structure, Lumberjack is able to improve its accuracy over a simple tree based learning method.

TTree is a tree based SMDP solver. It takes a generative model of the world and any abstract actions the user is able to supply. It uses them to build a policy and value function for the SMDP. The abstract SMDP that TTree forms internally is designed to make the policy effective before discretization is complete and the value function is accurate.

It should be noted that as yet we have not fed the output of the Lumberjack algorithm in as abstract actions to TTree. There is a format mismatch in that the output of Lumberjack is a Linked Decision Forest, and the input to TTree is a series of policies. The translation process we envisage is as follows: The input examples, in the form of state/action pairs, are passed through the linked decision forest. As well as recording frequency counts in the value leaves, the leaves of the root tree, we record them in the attribute leaves, the leaves of the other trees.

With this change we can now treat all trees in the forest as policies. Classification can start at the top of any tree. Attribute references are followed and returned from. Once a leaf of the tree we started in is reached, that leaf is treated as an value leaf, even if it is an attribute leaf. An action is picked randomly according to the distribution of actions stored in the leaf.

Aside from closing the gap between Lumberjack and TTree there are a number of other paths of future research opened by this thesis.

We noted in Chapter 5 that introducing bad splits in TTree can significantly reduce the quality of the abstract SMDP. Bad splits seem to be those that cut trajectories. One line of research involves modifying the splitting criterion so that splits that cut trajectories are minimized.

Along similar lines, if we are going to be trying to minimize splits that cut trajectories in TTree, it would be nice if the abstract actions supported this. It would be interesting to modify Lumberjack so that the splits introduced minimize the number of transitions cut in the optimal policy. This would introduce a bias similar to the one in HexQ (Hengst, 2002), but without the requirement for a constant variable ordering across the entire state space. It also gives a strong justification for that bias.

One big issue with TTree is the flat leaves. Unfortunately, these are also required for the proof of correctness. It would be worth investigating if this restriction can be relaxed while still maintaining correctness. Is it enough that, within a single abstract state, the actions have values with constant difference, rather than requiring them to be constant? (*i.e.* the Q function is a series of parallel planes within each leaf, but those planes are not necessarily level.)

It would be nice to have TTree do some intelligent pruning of the abstract actions it attempts to use. This might be achieved by either linking the application of abstract actions to features of the state space (*e.g.* only try the *Walk-Up-Stairs* abstract action if you see stairs) or by using adaptive sampling (*e.g.* adjust how many samples are taken for a particular action in a particular state depending upon whether the action is close to optimal in the current estimate).

Finally, TTree requires a Generative Model. This is a supervised learning problem. Can Lumberjack help with learning this generative model?

Bibliography

James Sacra Albus. *Brains, Behavior, and Robotics*. BYTE Publications, Peterborough, N.H., 1981. ISBN 0070009759.

Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference (ICML95)*, pages 30–37, San Mateo, 1995. Morgan Kaufmann.

R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

Michael Bowling and Manuela Veloso. An analysis of stochastic game theory for multiagent reinforcement learning. Technical Report CMU-CS-00-165, Computer Science Department, Carnegie Mellon University, 2000.

J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, Cambridge, MA, 1995. The MIT Press.

Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for large acyclic domains. In L. Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference (ICML96)*, San Mateo, CA, 1996. Morgan Kaufmann.

Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification And Regression Trees*. Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, CA, 1984.

Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 726–731, Sydney, Australia, 1991.

P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1993. Morgan Kaufmann.

Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1995.

Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML98)*, pages 118–126, Madison, WI, 1998. Morgan Kaufmann.

Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

Bruce L. Digney. Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. In *From Animals to Animats: Simulation of Adaptive Behavior (SAB96)*, 1996.

Bruce L. Digney. Learning hierarchical control structures for multiple tasks and changing environments. In *From Animals to Animats: Simulation of Adaptive Behavior (SAB98)*, 1998.

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

Geoffrey J. Gordon. Online fitted reinforcement learning. In *Value Function Approximation workshop at ML95*, 1995.

Milos Hauskrecht, Nicolas Meuleau, Craig Boutilier, Leslie Pack Kaelbling, and Thomas Dean. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 220–229, 1998.

Bernhard Hengst. Generating hierarchical structure in reinforcement learning from state variables. In Riichiro Mizoguchi and John K. Slaney, editors, *6th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000)*, volume 1886 of *Lecture Notes in Computer Science*. Springer, 2000. ISBN 3-540-67925-1.

Bernhard Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *International Conference on Machine Learning (ICML02)*, 2002.

Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Uncertainty in Artificial Intelligence (UAI-99)*, 1999.

Leslie Pack Kaelbling. Hierarchical reinforcement learning: Preliminary results. In *Machine Learning: Proceedings of the Tenth International Conference (ICML93)*, 1993.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Craig Alan Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1991.

Ron Kohavi. *Wrappers for Performance Enhancement and Oblivious Decision Graphs*. PhD thesis, Department of Computer Science, Stanford University, 1995.

Chun-Shin Lin and Hyongsuk Kim. CMAC-based adaptive critic self-learning control. *IEEE Transactions on Neural Networks*, 2(5):530–533, 1991.

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning: Proceedings of the Eleventh International Conference (ICML94)*, pages 157–163, San Mateo, 1994. Morgan Kaufmann.

Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, 1995.

Amy McGovern. *Autonomous Discovery Of Temporal Abstractions From Interaction With An Environment*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, 2002.

Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML01)*, pages 361–368, 2001.

W. Thomas Miller, III, Filson H. Glanz, and L. Gordon Kraft, III. CMAC: An associative neural network alternative to backpropagation. *Proceedings of the IEEE*, 78(10):1561–1567, October 1990.

Andrew W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 711–718. Morgan Kaufmann, 1994.

Andrew W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, 1993.

Andrew W. Moore, Leemon Baird, and Leslie Pack Kaelbling. Multi-value-functions: Efficient automatic action hierarchies for multiple goal mdps. In *International Joint Conference on Artificial Intelligence (IJCAI99)*, 1999.

Remi Munos and Andrew W. Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

Patrick M. Murphy and Michael J. Pazzani. Exploring the decision forest: An empirical investigation of Occam's razor in decision tree induction. *Journal of Artificial Intelligence Research*, 1:257–275, 1994.

Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, Computer Science, University of Waikato, Hamilton, New Zealand, 1996.

Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structures in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

Andrew Y. Ng and Michael Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference*, 2000.

J. Oliver and C. S. Wallace. Inferring decision graphs. Technical Report 91/170, Department of Computer Science, Monash University, November 1992.

Giulia Pagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–99, 1990.

Ron Parr. *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley, 1998.

Ron S Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Neural and Information Processing Systems (NIPS-98)*, volume 10. MIT Press, 1998.

J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. In *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior, Hawaii*, 1993.

Doina Precup. *Temporal Abstraction in Reinforcement Learning*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, 2000.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: the art of scientific computing*. Cambridge University Press, 1992.

Martin L. Puterman. *Markov Decision Processes : Discrete stochastic dynamic programming*. Wiley series in probability and mathematical statistics. Applied probability and statistics section. John Wiley & Sons, New York, 1994.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80(3):227–248, 1989.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1992.

Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of Statistics*, 11(2):416–431, 1983.

Satinder P. Singh. Transfer of learning across compositions of sequential tasks. In L. A. Birnbaum and G. C. Collins, editors, *Proceedings of the Eighth International Workshop on Machine Learning*, page 348352, San Mateo, CA, 1991. Morgan Kaufmann.

Satinder P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, Menlo Park, CA., 1992a. AAAI Press/MIT Press.

Satinder P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992b.

Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *NIPS-2000*, 2000.

J. A. Storer. Data compression via textual substitution. *Journal of the Association for Computing Machinery*, 29(4):928–951, 1982.

Malcolm Strens and Andrew Moore. Direct policy search using paired statistical tests. In *International Conference on Machine Learning (ICML 2001)*, 2001.

Richard S. Sutton. Generaliztion in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, pages 1038–1044. MIT Press, 1996.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

Richard S. Sutton, Doina Precup, and Satinder Singh. Intra-option learning about temporally abstract actions. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML98)*, pages 556–564, Madison, WI, 1998. Morgan Kaufmann.

Sebastian Thrun and A. Schwartz. Finding structure in reinforcement learning. In G. Tesauro and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 385–392, 1995.

P. E. Utgoff and C. E. Brodley. Linear machine decision trees. Tech. Report 91-10, University of Massachusetts, 1991.

William T. B. Uther and Manuela M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 769–774, Madison, WI, 1998.

C. S. Wallace and D. M. Boulton. An information measure for classification. *Computer Journal*, 11(2):185–194, 1968.

C. S. Wallace and J. D. Patrick. Coding decision trees. *Machine Learning*, 11:7–22, 1993.

Xin Wang and Thomas G. Dietterich. Stabilizing value function approximation with the BFBP algorithm. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

R. J. Williams and L. C. I. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Tech. report, College of Computer Science, Northeastern University, 1993.

Index